VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Modulární platforma pro prezentaci informací

Modular Platform for Presentation of Information

2012 Bc. Tomáš Cigánek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Cigánek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Modulární platforma pro prezentaci informací
Modular Platform for Presentation of Information

Zásady pro vypracování:

Cílem práce je vytvoření flexibilního systému pro prezentaci různých typů informací s využitím aktuálních technologií platformy Windows.

1. Analyzujte možnosti a požadavky a následně navrhněte systém, který by flexibilně umožňoval prezentaci různých typů informací, a to formou, která bude jakousi nadstavbou nad operační systém.
2. Systém bude umožňovat dynamické zapojování samostatných modulů, které budou pokrývat jednotlivé typy informací, např. poštovní schránka, RSS kanál, hodiny, počasí, apod. Tomuto musí odpovídat navržená flexibilní a otevřená architektura.
3. Při vývoji využijte moderní technologie pro vývoj uživatelského rozhraní, stejně jako možnosti dotykového ovládání.
4. Do systému integrujte sociální prvky, které budou sloužit především k analýze chování uživatelů a využívání uživatelských zkušeností.
5. Zhodnoťte výslednou aplikaci a možnosti jejího reálného využití.

Seznam doporučené odborné literatury:

Windows Phone 7 Metro Design UI Book - http://www.kurtbrockett.com/?p=71
WPF 4 Unleashed, Adam Nathan, 2010, ISBN: 978-0672331190

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Michal Radecký**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 31. března 2012 .....

Děkuji vedoucímu mé diplomové práce Ing. Michalovi Radeckému za jeho ochotu, čas, trpělivost a odbornou pomoc při řešení jednotlivých problému souvisejících s problematikou diplomové práce. Rovněž děkuji všem učitelům Vysoké školy báňské Technické univerzity Ostrava, že mne odborně provedli zvoleným oborem studia informatiky. Dále bych chtěl poděkovat i mé rodině nejen za finanční podporu, díky které jsem mohl absolvovat vysokou školu a psát tuto práci.

Abstrakt

Cílem této diplomové práce je vytvoření systému založeného na modulární platformě, který poskytuje rozhrání pro implementaci nových modulů. Ty budou uživateli prezentovat různé formy informací. Součástí práce bude návrh struktury pro práci s jednotlivými moduly. Systém bude obsahovat také návrh algoritmů, které budou realizovat inteligentní chování jednotlivých modulů v systému. Nedílnou součástí aplikace bude také zakomponování sociálních prvků v podobě uchovávání základních informací o jednotlivých modulech. Systém bude navržen tak, aby byl schopen jednoduše zakomponovat možnost dotykového ovládání.

Klíčová slova: Inteligentí mřížka, WPF, Modul, Master modul, Slave modul, Funkční modul, Testovací modul, Mřížka, Matice, Buňka, Sociální prvky

Abstract

The aim of this thesis is to create a system based on a modular platform which provides an interface for the implementation of new modules. They will present various forms of information to their users. A part of a work will design the structure of working with every single module. The system will also include the concept of algorithms, which will realize the intelligent behaviour of individual modules in the system. An integral part of the application will also be an incorporation of social elements in the form of retention of basic information about individual modules. The system will be designed to be able easily incorporate the possibility of stylus control.

Keywords: Clever grid, WPF, Module, Master module, Slave module, Functional module, Test module, Grid, Matrix, Cell, Social elements

Seznam použitých zkratek a symbolů

WCF	–	Windows Communication Foundation
WPF	–	Windows Presentation Foundation
SOA	–	Servisně orientovaná aplikace
.NET	–	Dot Network
MVVM	–	Model View ViewModel
RSS	–	Rich Site Summary

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Hlavním cílem této práce by mělo být vytvoření uživatelsky přívětivého systému, který je založen na modulární platformě. Díky tomu je možno do něho zapojovat moduly, které se v něm umí chovat podle předem daných pravidel. Jednotlivé moduly slouží pro prezentaci různých forem informací. Je zde navržena struktura pro práci a ukotvení jednotlivých modulů, která se nazývá **inteligentní mřížka**. Ta obsahuje implementaci vlastních algoritmů, které realizují inteligentní funkce mřížky. Dále pak hlídají a řídí chování při manipulaci s jednotlivými moduly. Mezi tyto algoritmy patří například vkládání nových modulů, jejich přemisťování, hledání volných pozic či přepínání mezi stavy modulů. Aplikace umí také uchovávat informace o sociálních aspektech. Tato data pak zpracovávat a vyhodnocovat.

Práce je rozdělena do čtyř logických částí. Po přečtení jednotlivých kapitol by měl být čtenář postupně zaváděn do hlubší problematiky problému a na konci by měl být schopen plně se systémem spolupracovat, detailně pochopit řešené problémy a sám si například implementovat nový modul.

Obsahem první části je popis existujících řešení dané problematiky, cíle práce a inspirace k její realizaci.

Druhá hlavní kapitola se zabývá obecným pohledem na systém a jeho bloky. Jsou zde popsány základní informace o aplikaci a jejich části, z kterých se skládá. Obsahem je i abstraktní pohled na její architekturu.

Následující kapitola je stěžejní a obsahuje popis fungování jednotlivých inteligentních algoritmů, struktury aplikace, detailní informace o implementovaných modulech a jejich stavech.

Poslední část se zabývá práci se sociálními prvky, jejich zpracováním a návazností aplikace na možnosti jejího ovládání pomocí technologie multidoteků.

V závěru je celá práce zhodnocena a jsou zde popsány možnosti jejího dalšího vývoje, jako je například optimalizace použitých algoritmů, další zpracování a vyhodnocování sociálního chování aplikace.

2 Charakteristika problematiky

V dnešní době je velmi populární využívat ovládání různých typů zařízení pomocí dotyků prstem. Mezi tato zařízení patří mobilní telefony, tablety a v současné době i samotné počítače. U nich je to především díky podpoře operačních systémů. Jedním z nich je v současné době velmi hojně používaný **Windows 7**. Ten se však ve většině případů stále ovládá standardním způsobem, protože jeho uživatelské rozhraní se nijak zvláště multidotekům nepřizpůsobilo. Tento problém je však vyřešen v následovníkovi toho systému. Ten se jmenuje **Windows 8** a je inspirací pro tuto práci, jelikož je založen na modulární platformě. Poskytuje také rozhraní, které může být využito při zakomponování technologie multidoteků a umí pracovat se sociálními prvky. Dalším aspektem je i jednoduchost uživatelského rozhraní.

2.1 Existující řešení Windows 8 a Metro

Windows 8 je nástupcem **Windows 7**. Novinky technologie systému **Windows 8** jsou založeny kromě požadavku na jednoduchost vzhledu hlavně na využití multidoteků. Jelikož v dnešní době velké množství uživatelů využívá populární tablety, je tato technologie především vhodná pro ně. Jednotlivé panely systému jsou "živé", což znamená, že uživatel má neustále aktuální informace, jako jsou například přehled kurzů, televizní seznam či aktuální stav počasí nebo čas. Poskytuje tedy všechny potřebné a důležité informace. Aplikace mezi sebou komunikují a společně spolupracují. Systém **Windows 8** komunikuje se službou *Windows Store*. Ta poskytuje tisíce užitečných aplikací ke stažení. Je možno si je před zakoupením také vyzkoušet. Po zakoupení je možno je využívat až na pěti zařízeních. Pokud je uživatel přihlášen pomocí Microsoft účtu kdekoliv a na jakémkoliv zařízení (notebook, tablet, počítač, . . .), je automaticky nastaven **Metro** styl daného uživatele. To znamená, že si systém stáhne nejen nastavený vzhled daného uživatele, ale například rozložení jednotlivých aplikací, spuštěné a prohlížené webové stránky a další spoustu užitečných věcí. **Windows 8** poskytuje jednotný vzhled a to právě díky **Metro** stylu.

Window 8 zavádí vzhled nového uživatelského rozhraní, který nese název **Metro** styl. Tento styl je postaven na jednoduchosti, intuitivnosti. Využívá čisté typografie. Pro příjemnou práci uživatele používá mimo jiné také velké množství animací. Je v interakci s uživatelem. Je zábavný a využívá jednoduchých pohybů. Uživatel se díky tomuto stylu snadno v systému orientuje a může si jej přizpůsobovat. Umožňuje se pohybovat pomocí myši, klávesnice nebo prstem. Poskytuje snadné rozhraní pro vývoj aplikací v tomto stylu.[1]

2.2 Stručná charakteristika řešení Inteligentní mřížky

2.2.1 Inspirace

Jelikož se věnuji vývoji desktopových aplikací a webové systémy mě nikdy nějak moc nepřitahovaly, tak jasnou motivací bylo implementovat systém, který bude využívat

Obrázek 1: Windows 8

technologie *WPF*. Tuto skutečnost ovlivnil fakt, že tuto technologii používám i při své práci a chtěl jsem se v ní zdokonalit. Postupně jsem tedy začal navrhovat systém, který se zabývá chováním jistých objektů v mřížce. Nejprve jsem myslel, že půjde pouze o změnu pozice daných objektů. Postupným zkoumáním jsem však došel i ke změně velikosti jednotlivých objektů, což mělo za následek nárůst velkého množství problému z toho vyplývajících. Tudíž se z celkem jednoduchého systému stal systém vcelku složitý.

Jednou z hlavních inspirací pro zanoření se do problematiky chování objektů v mřížce byla již zmíněná technologie **Windows 8**. Jak je vidět na obrázku 1, jsou jednotlivé aplikace umístěny v jakési pomyslné mřížce, což vedlo k realizaci takovéto mřížky. Ta by mohla být používána třeba i v systému **Windows 8**. Dalším důvodem pak byl návrh aplikace s jednoduchým uživatelským rozhraním jako je na obrázku 1.

Jelikož vzhled systému **Windows 8** je založen na co největší možné jednoduchosti, byla snaha se v něm inspirovat a vzhled aplikace navrhnout podobným způsobem jako je použití **Metro** stylu. Tedy jednoduchost použitých barev, obrázků, tlačítek či ikon.

2.2.2 Cíle

Systém bude mít jednotný vzhled a jednoduché uživatelské rozhraní. Dalším hlavním požadavkem je, aby obsahoval mřížku, do které bude možno připojovat a odpojovat jednotlivé moduly. Ta bude konfigurovatelná. Uživatel bude moci nastavit její velikost, tedy počet sloupců a řádků mřížky. Systém bude zajisté obsahovat několik modulů, na kterých bude možno testovat jejich chování. Dále bude obsahovat několik konkrétních modulů, které budou poskytovat uživateli nějakou funkcionalitu. To znamená, že moduly budou uživateli prezentovat různé druhy informací. Mezi ně patří:

- aktuální čas
- čtečka *RSS* kanálů
- fotky(obrázky)

Jednotlivé moduly se budou v mřížce chovat jako "živé" a uživatel bude moci sledovat jejich chování a dále s nimi pracovat. Mřížka mu poskytuje množství různých funkcí pro práci s nimi.

- Změna velikosti
- Změna pozice
- Stavy
- Změna stavu a automatické přepínání mezi nimi
- Optimální umístění

Moduly mohou také na tyto úkony reagovat a podle určitých algoritmů se budou moci sami přemisťovat nebo měnit svou velikost a stavy. Díky těmto funkcím se mřížka dále nazývá jako **inteligentní**. Systém bude mít možnost poskytovat rozhraní pro vývoj nových modulů. Pokud si uživatel bude chtít vytvořit svůj vlastní modul, bude mu poskytnuto příslušné rozhraní. Na základě něj si může vytvořit svůj vlastní modul a nezávazně ho přidat do mřížky. Pro usnadnění práce uživatele, bude systém evidovat chování jednotlivých uživatelů. V závislosti na konfiguraci bude možno evidovat data o pozicích a velikostech jednotlivých modulů. Na základě těchto informací budou například moduly v mřížce automaticky umisťovány dle nejpoužívanější pozice či velikosti.

Projekt bude splňovat požadavky na vhodnou strukturu a architekturu.

3 Inteligentní mřížka

Rozvržení aplikace pro práci s jednotlivými moduly je založeno na jakési tabulce neboli mřížce. Ta je definována určitým počtem řádků a sloupců. Velikosti jednotlivých složek je možno uživatelsky definovat. Na základě těchto informací systém automaticky rozloží velikosti jednotlivých buněk. Tedy jejich výšku a šířku. Buňkou je nazýván prvek, který je definovaný svým indexem. Tedy pokud existuje mřížka, která bude obsahovat čtyři řádky a čtyři sloupce, tak buňkou můžeme nazvat prvek na pozici <2, 3>. Tato buňka je na třetím řádku a čtvrtém sloupci. Jednotlivé indexy pro řádky a sloupce začínají nulou.

Inteligentní mřížka je tedy mřížka o určitém počtu sloupců a řádků, do které je možno vkládat či odebírat moduly. Jejím hlavním úkolem je hlídat a řídit chování jednotlivých modulů. Na obrázku 2 je vidět aplikaci v konfiguraci, kdy je nastaven počet řádku na deset a sloupců na patnáct.

3.1 Možnosti chování v mřížce

Algoritmy, se kterými spolupracuje **inteligentní mřížka**, umí rozpoznávat a zpracovávat dvě základní operace. Tyto operace jsou **přesun** a **změna velikosti** modulů. Pro každou operaci se dále vyhodnocuje, jestli jde o ***master*** nebo ***slave*** modul. ***Master*** modul je prvek, se kterým je uživatel v přímé interakci a přímo s ním pracuje. ***Slave*** modul je prvek, který je ovlivněn na základě toho, že uživatel pracuje s ***master*** modulem. Typickým příkladem je situace, kdy uživatel zvětší velikost ***master*** modulu tak, že jeho nová velikost zasahuje až do oblasti, kde je umístěný jiný modul. Tento modul se pak nazývá ***slave*** modulem.

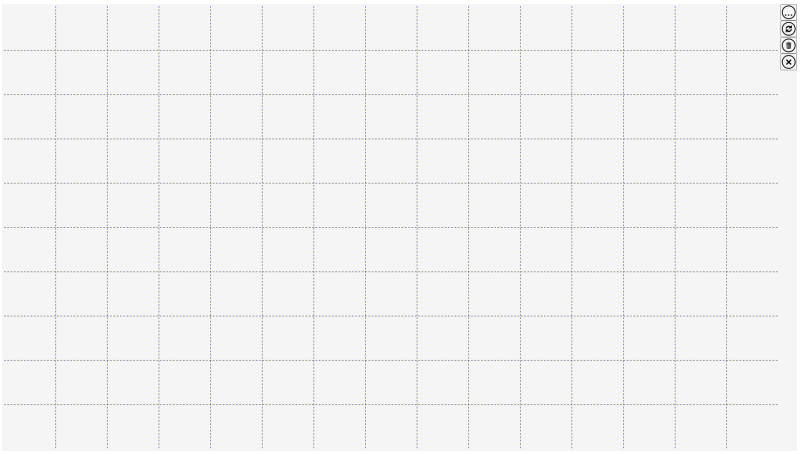
Inteligentní mřížka zpracovává jednotlivé typy chování. Zajišťuje přesuny a změny velikostí jednotlivých typů modulů. Pokud vyhodnotí, že s vybraným modulem nelze provést některou z operací, uživateli tento krok není povolen. Typickým příkladem takovéto situace je, kdy uživatel chce zvětšit ***master*** modul na takovou velikost, že některý z ***slave*** modulů už nelze dále přemístit či zmenšit.

3.1.1 Přesun

První ze základních operací je přesun. Tuto funkci může uživatel využít, pokud mu nevyhovuje současná pozice modulu nebo pozice, kam byl modul vložen po jeho přidání. Pokud chce tedy uživatel změnit pozici vybraného modulu, musí jej přepnout do módu pro přesun. Poté mu je umožněno daný modul přesouvat libovolně v rozmezí **inteligentní mřížky**. Za tyto hranice není povolenou modul přesouvat.

3.1.2 Změna velikosti

Druhou operací se nazývá změna velikosti modulu. Pokud uživatel není spokojen s aktuální velikostí nebo s velikostí, které byla automaticky nastavena po vložení do **inteligentní mřížky**, je možno tuto velikost taktéž měnit. Uživatel musí přepnout modul do módu pro změnu velikosti modulu a poté je mu umožněno tuto změnu provést. Lze ji provádět jak



Obrázek 2: Prázdná mřížka

vertikálně tak i horizontálně. Změny velikosti lze provádět pouze na úrovni **inteligentní mřížky**, za tuto oblast systém změnu velikosti nepovolí.

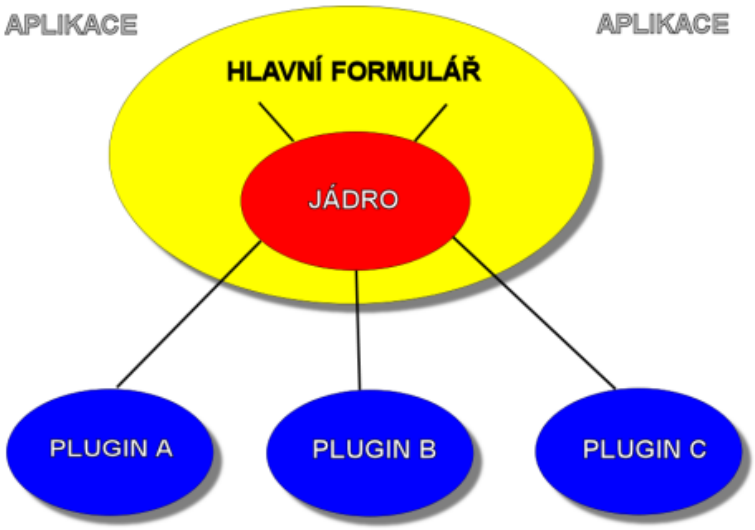
V závislosti na velikosti mřížky se dané moduly přepínají mezi jednotlivými stavy, které daný modul poskytuje. Každý z modulů implementuje takzvaný **minimální stav**. Tento stav reprezentuje nejmenší možnou velikost vybraného modulu. Pokud tedy bude uživatel chtít zmenšit výšku nebo šířku modulu pod tuto úroveň minimálního stavu, **inteligentní mřížka** mu tuto operaci nepovolí.

3.2 Obecná práce s modulem

Modulem se rozumí aplikace, která však není samostatně spustitelná. V některých konkrétních implementacích se jedná o aplikaci, která reprezentuje různé druhy informací. Tento modul se nazývá **funkční modul**. Druhým typem modulu je **testovací modul**. Tento slouží pouze jako prvek pro testování chování v **inteligentní mřížce**.

Aplikace dále umožňuje několik operací, které lze s vybraným modulem provádět.

- **Vložení vybraného modulu do mřížky** - Na základě použitého algoritmu se modul umístí na nejoptimálnější volnou pozici v mřížce.
- **Hromadného připojení** - Na základě vybrané cesty k adresáři systém vyhledá moduly a ty pak v náhodném pořadí do mřížky přidá. Přitom se opět pro každý modul dohledá nejvhodnější volné místo v mřížce. Počet takto náhodně vybraných modulů si může uživatel sám konfigurovat.
- **Hromadného odstranění všech modulů** - Odebere všechny moduly připojené v mřížce.



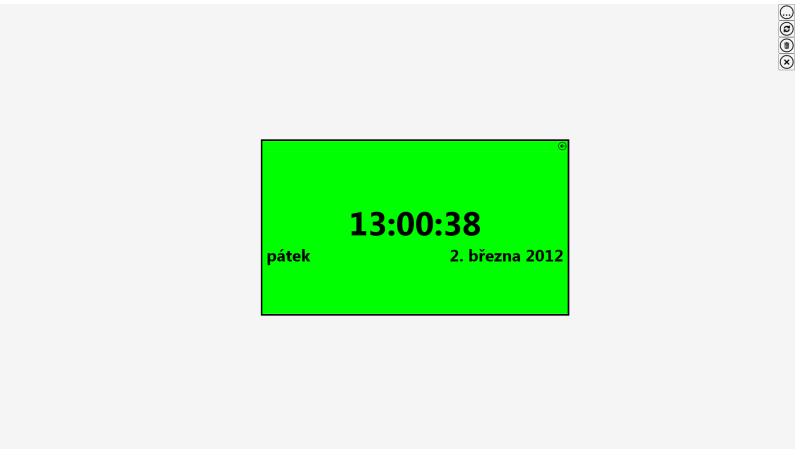
Obrázek 3: Modulární platforma

3.2.1 Modulární architektura

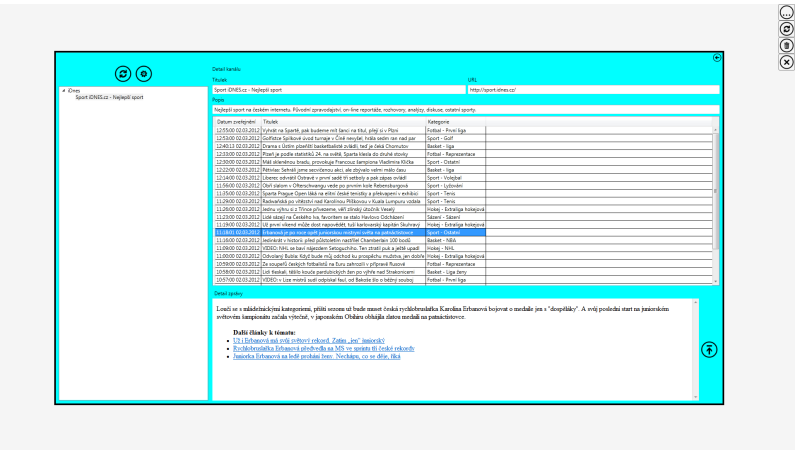
Systém je navržen na základě **modulární architektury**. Tou je myšlena aplikace, která umožňuje komunikaci s externími prvky. Těmito prvky se rozumí takzvané "**pluginy**" neboli moduly. To jsou samostatné části, které nejsou nebo nemusí být součástí celé aplikace. Jádro aplikace tvoří prostředníka mezi vlastní aplikací a těmito moduly. Tato architektura je zachycena na obrázku 3. Tyto moduly musí splňovat určitá pravidla, která jsou předem určena. Je to z důvodu, aby jádro bylo schopno modul rozpoznat a na základě těchto pravidel s ním správně komunikovat. Takováto aplikace umožňuje za běhu jednotlivé moduly připojovat či odpojovat.[2]

3.3 Použité moduly

Existují dva typy modulů a to **testovací** a **funkční**. Jelikož implementace **testovacích modulů** není příliš náročná, jsou implementovány čtyři moduly. Realizace **funkčních modulů** je výrazně náročnější a jsou vytvořeny pouze tři základní. Názvy jsou odvozeny od jejich funkcionality. Mezi ně patří moduly s názvy **Time**, **RssReader** a **PhotoDirectory**. Jelikož **testovací moduly** nemají téměř žádnou funkčnost, tak jsou použity pro jejich odlišení názvy barev pozadí. **Testovací moduly** tedy nesou názvy **Red**, **Orange**, **Gray** a **Blue**.



Obrázek 4: Modul *Time*



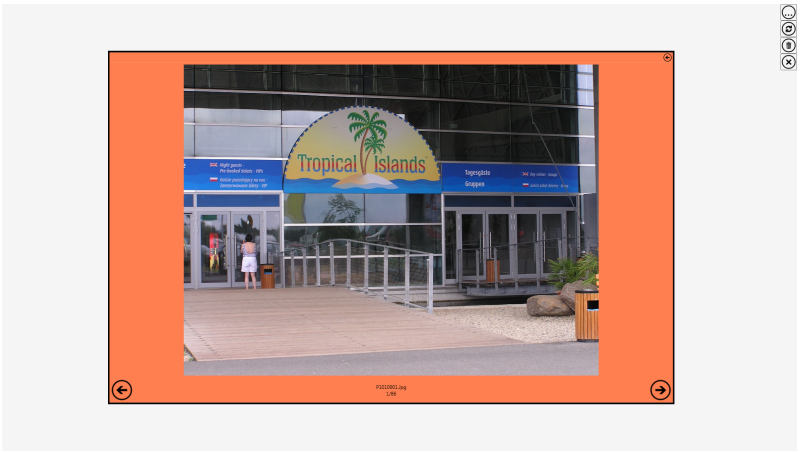
Obrázek 5: Modul *RssReader*

3.3.1 Modul Time

Tento modul má ze všech **funkčních modulů** nejtriviálnější funkci. Jedná se pouze o prezentaci informací týkajících se aktuálního času a data. Úplný vzhled je možno vidět na obrázku 4.

3.3.2 Modul RssReader

Z názvu je již patrné, že se tento modul stará především o prezentaci informací prostřednictvím *RSS* kanálů. *RSS* kanál je informační zdroj, který poskytuje aktuální informace ve formátu *XML*. Tyto zprávy musí splňovat jistá pravidla, která je nutno dodržovat. Modul **RssReader** je schopen tato data přečíst a prezentovat je uživatelům v čitelné podobě. Vzhled modulu je možno vidět na obrázku 5.



Obrázek 6: Modul *PhotoDirectory*

3.3.3 Modul PhotoDirectory

Posledním z **funkčních modulů** je **PhotoDirectory**. Tento modul slouží pro prezentaci obrázků či fotek. Uživatel si může zvolit adresář, který obsahuje obrázky ve formátech *JPG*, *GIF*, *BMP*, *TIF* nebo *PNG*. Po zapnutí modulu se z vybraného adresáře načtou obrázky příslušných formátů. Uživatel si je pak může postupně prohlížet. Modul je vidět na obrázku 6.

3.3.4 Testovací modul

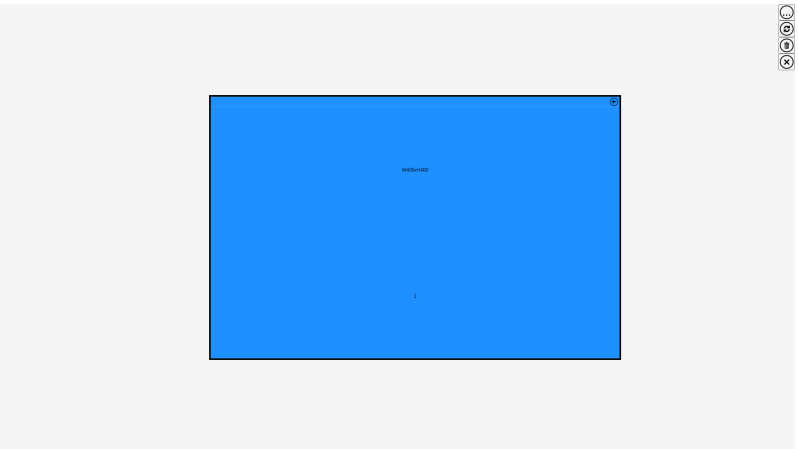
Jelikož všechny **testovací moduly** mají stejný vzhled i funkcionalitu a liší se pouze v barvě pozadí, tak je zde prezentován pouze vybraný **testovací modul** s názvem **Blue**. Na pozadí se pouze prezentuje pořadí, ve kterém byl daný modul vložen do **inteligentní mřížky** a minimální velikost aktuálního stavu, ve kterém se modul nachází. Jak modul vypadá je vidět na obrázku 7.

3.4 Technologie a postupy pro nejvhodnější řešení dané problematiky

3.4.1 Dědičnost a návrhové vzory

Hlavním smyslem návrhu je snaha navrhnout a použít pro realizaci projektu nejvhodnější možné technologie. Již při počátečních pokusech bylo zjištěno, že chování některých komponent projektu je totožné nebo velmi podobné. Jedná se především o jednotlivé moduly. Proto byla snaha využívat možnosti dědičnosti mezi objekty.

Jednou z dalších ověřených technologií jsou návrhové vzory. Jelikož je tato technologie vřele doporučována, tak jsou vybrané vzory implementovány i v této práci. V průběhu realizace se objevilo několik situací a problémů, které přímo vybízely k jejich využití.



Obrázek 7: Testovací modul

3.4.2 Rozložení struktury projektu

Při realizaci projektu docházelo k velkému přírůstku nových objektů, které byly využívány. V určité situaci jich bylo už natolik hodně, že orientace mezi nimi byla matoucí. Proto bylo nutné logicky rozčlenit strukturu projektu. Jelikož byly některé funkce využívány stále častěji, bylo nutné si je oddělit do jednotlivých knihoven. V některých situacích to byla dokonce nutnost.

K hlavnímu členění došlo v situaci, kdy se začalo řešit **sociální chování** v **inteligentní mřížce**. Jelikož by odchytávání informací o chování jednotlivých modulů mělo být nezávislé a technologie použitá pro komunikaci si to přímo vynucovala, bylo nutné zpracovávání těchto informací realizovat v samostatně odděleném projektu.

V důsledku toho vznikly dva projekty. Jeden pro zpracování informací o chování modulů v **inteligentní mřížce**, který plní funkci serveru. Druhým a tím hlavním je projekt, který se zabývá řízením chování a zprávou modulů.

4 Algoritmy chování inteligentní mřížky

4.1 Implementace inteligentní mřížky

Základní komponentou, která obsahuje implementaci **inteligentní mřížky** je *CG.xaml*, která je umístěna v projektu *CleverGridApplication*. Veškerá uživatelská rozhraní, která jsou použita v aplikaci, jsou implementovaná pomocí technologie *WPF*. Tato technologie nabízí velké množství komponent pro práci a prezentaci různých forem informací. Nejvhodnějším prvkem z poskytované kolekce se jevila komponenta, která se nazývá *Grid*. Tato komponenta nabízí velké množství užitečných funkcí, které byly při vývoji potřeba a proto byl *Grid* zvolen jako jádro pro implementaci **inteligentní mřížky**. Základním požadavkem je možnost dynamického nastavování počtu řádků a sloupců. Tuto možnost *Grid* poskytuje. Jelikož však má být tato konfigurace dynamická, nelze ji vložit přímo do *xaml* souboru. Proto je nutné načítání sloupců a řádků dělat přímo v kódu. Informace o velikosti mřížky může uživatel nastavovat v konfiguračním souboru aplikace. Klíč *CountColumns* definuje, kolik bude mít mřížka sloupců. *CountRows* určuje počet řádků. Na základě této konfigurace pak systém při spuštění aplikace inicializuje mřížku a nastaví jí příslušnou velikost.

Součástí jedné nebo více buněk v mřížce může být modul. Komponenta *Grid* tuto možnost podporuje. Je potřeba, aby modul bylo možno přesunout nebo mu změnit velikost. Proto bylo hledáno řešení, jak tento problém vyřešit. Jedním z možných, bylo využít techniky *DragAndDrop*. Tento způsob je založen na registraci a odchytávání mnoha událostí nad konkrétním objektem. V tomto případě by to byla komponenta *Grid*. Mezi tyto události patří *DragOver*, *DragEnter*, *DragDrop* a další. Po implementaci této metody se však objevil zásadní problém. Tuto techniku lze použít pouze pro přesun. Dalším omezením bylo, že se používalo pozicování konkrétních objektů. A to se nezdálo jako vhodné řešení. Proto bylo nutné, hledat řešení jiné. Alternativou bylo použití komponenty *Canvas*, na které se dá s objekty jednoduše pracovat. Tedy **zvětšovat**, **zmenšovat**, **otáčet** či **přesouvat**. Do této komponenty byl vložen objekt *Grid*. To je zobrazeno ve výpisu 1.

```
<Canvas x:Name="canvas" VerticalAlignment="Stretch" HorizontalAlignment="Stretch" Margin="5">
  <Grid x:Name="gCleverGrid" Width="{Binding␣ElementName=canvas,␣Path=ActualWidth}"
      Height="{Binding␣ElementName=canvas,␣Path=ActualHeight}" ShowGridLines="True"/>
</Canvas>
```

Výpis 1: *Grid* v *Canvasu*

4.1.1 Šablony a styly

Jelikož je pro operace s moduly použita komponenta *Canvas* je nutné, aby obsahem buňky mřížky byla komponenta *ContentControl*. Tyto kompnenty spolu jednoduše spolupracují. *ContentControl* jednoduše zaobaluje modul a umožňuje mu jednoduše se pohybovat na komponentě *Canvas*. *ContentControlu* je nutné pouze vytvořit dva speciální **styly**. Jeden pro **přesun** a druhý pro **změnu pozice** aktuálního modulu. Pro realizaci je potřeba využít dalšího objektu. Ten se nazývá *Thumb*. Pro každou z těchto operací je vytvořena zvláštní třída, *MoveThumb* pro **přesun** a *ResizeThumb* pro **změnu velikosti** objektu v mřížce. Každá

z těchto tříd musí dědit z objektu *Thumb*. Objekt *Thumb* umí spolupracovat s komponentou *Canvas*. Ta obsahuje událost *DragDelta*, která je vyvolána vždy, pokud na komponentě *Canvas* dojde ke změně pozice myši. Tuto událost musí registrovat obě odvozené třídy.

Aby byl schopen objekt *MoveThumb* odchytnou událost *DragDelta* je nutné, aby se *ContentControlu* nastavil příslušný styl. Jeho realizace je tvořena ze dvou částí. V první se vytváří takzvaná ***ControlTemlate* šablona**, která je na výpisu 2.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http :// schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:thumbs="clr-namespace:CleverGridApplication.CleverGrid.Thumbs">
  <ControlTemplate x:Key="MoveThumbTemplate" TargetType="{x:Type thumbs:MoveThumb}">
    <Rectangle Fill="Transparent"/>
  </ControlTemplate>
</ResourceDictionary>
```

Výpis 2: Šablona *MoveThumbTemplate*

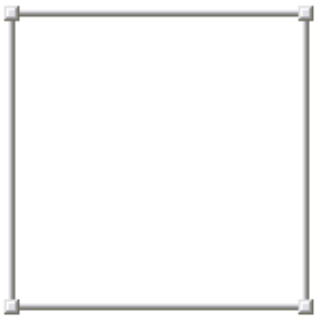
ControlTemplate je typu *MoveThumb*, což je třída pro přesun, která byla odvozena od objektu *Thumb*. Obsahem této šablony je obdélník, který bude průhledný. Tato jednoduchá šablona pak bude součástí hlavního stylu pro přesun po komponentě *Canvas*. Definice tohoto stylu je na výpisu 3.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http :// schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:thumbs="clr-namespace:CleverGridApplication.CleverGrid.Thumbs">
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="MoveThumb.xaml"/>
    <ResourceDictionary Source="ResizeThumb.xaml"/>
  </ResourceDictionary.MergedDictionaries>
  <Style x:Key="PresenterItemTemplateMove" TargetType="ContentControl">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="ContentControl">
          <Grid DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}"
              VerticalAlignment="Stretch" HorizontalAlignment="Stretch" >
            <thumbs:MoveThumb Template="{StaticResource MoveThumbTemplate}" Cursor="
                SizeAll"/>
            <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
          </Grid>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

Výpis 3: Styl *PresenterItemTemplateMove*

Tento styl je pak přiřazen *ContentControlu* a jmenuje se **PresenterItemTemplateMove**. Dále se mu pak definuje výsledná šablona. Její součástí je předchozí šablona a objekt *ContentPresenter*, který prezentuje obsah již konkrétně použitého *ContentControlu*.

Stejně jako odchycení události pro změnu pozice, tak i pro změnu velikosti, je nutno definovat opět několik šablon a styl pro *ContentControl*. Základní šablona slouží pro



Obrázek 8: Posuvníky pro změnu velikosti

definici hran, prostřednictvím kterých se bude s *ContentControlem* posouvat. Definuje čtyři styly pro hranu každé strany. Jedná se o hrany pro změnu velikosti ve směru **nahoru**, **dolů**, **vpravo** a **vlevo**. Dále pak ještě čtyři styly pro jednotlivé rohy, které slouží pro pohyby **vpravo dolů**, **vlevo dolů**, **vlevo nahoru** a **vpravo nahoru**. Takovýto rámeček je na obrázku 8.

Konkrétní styl pro změnu velikosti vpravo je na výpisu 4.

```
<Style x:Key="sSizeWERight" TargetType="{x:Type␣thumbs:ResizeThumb}">
  <Setter Property="Width" Value="3" />
  <Setter Property="Cursor" Value="SizeWE" />
  <Setter Property="Margin" Value="0,0,−4,0" />
  <Setter Property="VerticalAlignment" Value="Stretch" />
  <Setter Property="HorizontalAlignment" Value="Right" />
</Style>
```

Výpis 4: Styl pro změnu velikosti pravou hranou

Tento styl je opět určen pro dříve definovanou třídu *ResizeThumb*, která je odvozena od objektu *Thumb*. Definuje šířku čáry a její zarovnání. Výšku není nutno definovat, protože se odvíjí od výšky *ContentControlu*, který má tento styl nastaven. Dalšími důležitými vlastnostmi je nastavení správného kursoru. Tedy ikony, která reprezentuje pozici myši. V okamžiku, kdy je myš na pozici pravé hrany, která mění velikost, ikona myši se změní na takovou, která odpovídá změně velikosti vpravo. Podobně jsou definovány styly pro jednotlivé rohy. Na výpisu 5 je vidět styl pro posun vpravo dolů.

```
<Style x:Key="SizeNWSEBottomRight" TargetType="{x:Type␣thumbs:ResizeThumb}">
  <Setter Property="Height" Value="7" />
  <Setter Property="Width" Value="7" />
  <Setter Property="Cursor" Value="SizeNWSE" />
  <Setter Property="Margin" Value="0,0,−6,−6" />
  <Setter Property="VerticalAlignment" Value="Bottom" />
  <Setter Property="HorizontalAlignment" Value="Right" />
</Style>
```

Výpis 5: Styl pro změnu velikosti pravým dolním rohem

Nastavení vlastností je velmi podobné jako u předchozího stylu. Jediným rozdílem pro rohy je, že se zde definuje jak výška, tak šířka. Výsledné styly se vloží do jednoduché šablony na výpisu 6.

```
<ControlTemplate x:Key="ResizeThumbTemplate" TargetType="{x:Type Control}">
  <Grid>
    <thumbs:ResizeThumb Style="{StaticResource sSizeNSTop}"/>
    <thumbs:ResizeThumb Style="{StaticResource sSizeNSBottom}"/>
    <thumbs:ResizeThumb Style="{StaticResource sSizeWELeft}"/>
    <thumbs:ResizeThumb Style="{StaticResource sSizeWERight}"/>

    <thumbs:ResizeThumb Style="{StaticResource sSizeNWSETopLeft}"/>
    <thumbs:ResizeThumb Style="{StaticResource sSizeNESWTopRight}"/>
    <thumbs:ResizeThumb Style="{StaticResource sSizeNESWBottomLeft}"/>
    <thumbs:ResizeThumb Style="{StaticResource SizeNWSEBottomRight}"/>
  </Grid>
</ControlTemplate>
```

Výpis 6: Šablona *ResizeThumbTemplate*

Takto připravená šablona je pak součástí stylu, který je nastaven příslušnému *ContentControlu*. Výsledný styl pro změnu velikosti je možno vidět na výpisu 7.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http :// schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:thumbs="clr-namespace:CleverGridApplication.CleverGrid.Thumbs">
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="MoveThumb.xaml"/>
    <ResourceDictionary Source="ResizeThumb.xaml"/>
  </ResourceDictionary.MergedDictionaries>
  <Style x:Key="PresenterItemTemplateResize" TargetType="ContentControl">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="ContentControl">
          <Grid DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}"
              VerticalAlignment="Stretch" HorizontalAlignment="Stretch">
            <Control Name="ResizeThumb" Template="{StaticResource ResizeThumbTemplate}"
                />
            <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
          </Grid>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

Výpis 7: Styl *PresenterItemTemplateResize*

Tento styl se jmenuje *PresenterItemTemplateResize*. V něm se obdobně definuje šablona *ControlTemplate*. Její součástí je šablona *ResizeThumbTemplate* a objekt *ContentPresenter*, který prezentuje opět obsah již konkrétně použitého *ContentControlu*.

Pokud tedy tyto styly přiřadíme *ContentContolu*, je možno odchytit a dále zpracovat událost *DragDelta*. Na základě použité šablony a vyvolání této události je možno zjistit,

zda došlo ke změně pozice či velikosti konkrétního modulu, se kterým bylo pracováno. Algoritmus, který řeší problém přesunu modulu v **inteligentní mřížce**, je popsán v kapitole 4.4.1. Algoritmus, který řeší problém změny velikosti modulů v **inteligentní mřížce**, je pak popsán v kapitole 4.4.2.

U obou výsledných šablon *MoveThumbTemplate* a *ResizeThumbTemplate* bylo vidět v jejich definici, že obsahují *Grid*, ve kterém jsou další objekty. Tomuto *Gridu* se nastavoval vždy *DataContext* na *TemplatedParent*. Je to proto, že tento *DataContext* se propisuje až do nejnižší úrovně. Tedy pokud *Grid* obsahoval dvě položky, tak se tento *DataContext* nastavil i těmto položkám. Díky tomu je po odchycení události *DragDelta* možno zjistit, o jaký *ContentControl* se jedná.[3]

4.1.2 CGHelper a struktura organizace buněk inteligentní mřížky

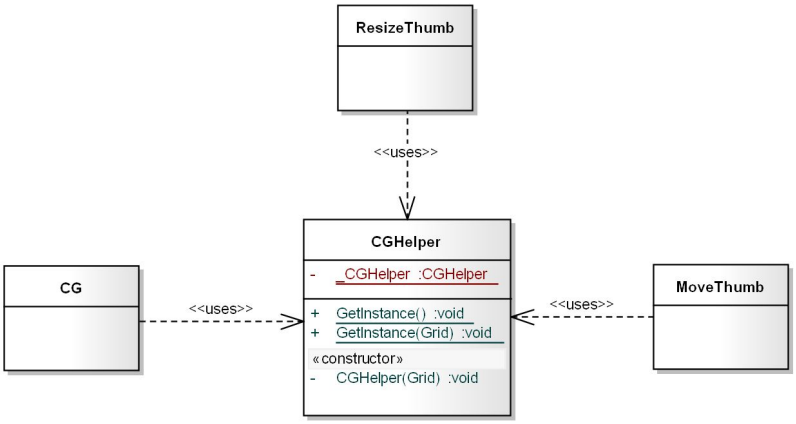
Hlavním jádrem **inteligentní mřížky** je třída *CGHelper*. V ní jsou implementovány všechny důležité funkce pro práci s moduly. Řídí také správné chování ***master*** modulů a jejich umístění v mřížce. S instancí této třídy úzce spolupracují obě třídy definované v kapitole 3.1. Jedná se o objekty *ResizeThumb* a *MoveThumb*, které slouží pro odchycení přesunu modulu nebo změnu jeho velikosti. Dále ji využívá i vlastní *Grid*, tedy třída *CG.cs*, ve které je komponenta *Grid* umístěna. Tato třída se dále stará i o správné chování a umisťování ***slave*** modulů, které bylo nutno přemístit nebo změnit jejich velikosti na základě práce s některým s ***master*** modulů. Mezi další z mnoha funkcí patří zjištění pozice umístění konkrétního modulu v mřížce nebo odchycení události o změně stylu pro konkrétní úkon. Tím je myšleno, že konkrétnímu modulu nastaví dle odchycené události správnou šablonu pro změnu velikosti nebo pozice modulu, se kterým uživatel pracuje.

Jelikož si konkrétní instance objektů *ResizeThumb* a *MoveThumb* drží systém někde na pozadí, nebylo možné po odchycení události změny pozice nebo velikosti modulu přistupovat k této stěžejní třídě. Tato situace přímo vybízí k použití **návrhového vzoru**. Tento vzor se jmenuje **Singleton**. Jeho popis a schéma je v příloze B.1.

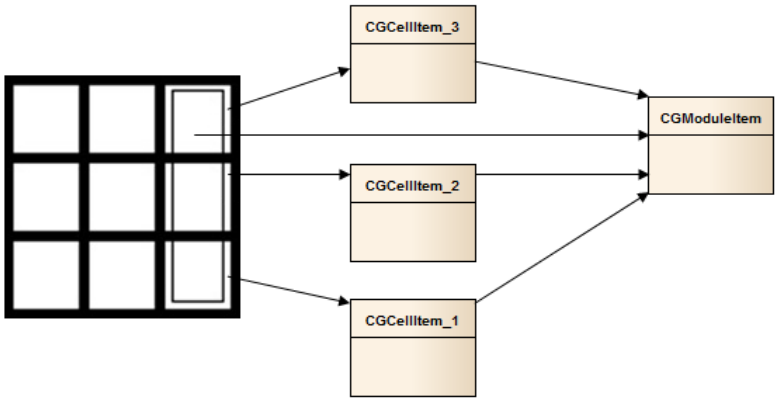
Díky tomuto vzoru nyní není problém k této třídě přistupovat. Dalším důvodem využití tohoto vzoru bylo i to, že si třída **CGHelper** drží uvnitř spoustu informací. Tyto informace jsou jednotné pro všechny objekty, které k této třídě přistupují. Proto by bylo velmi složité a možná i nemožné si je udržovat, pokud by si každý objekt držel svou vlastní instanci této třídy. Mezi tyto objekty patří například již dříve zmiňovaný počet sloupců a řádků. Vlastní řešení toho vzoru v aplikaci je zobrazeno na diagramu, který je na obrázku 9.

CGHelper využívají třídy *ResizeThumb*, *MoveThumb* a *CG*. Základem je, že musí existovat nějaká privátní statická proměnná, která je typu *CGHelper*. Další důležitou podmínkou, pro správnou implementaci toho vzoru je, že tato třída musí mít pouze privátní konstruktor. Pro získání jediné instance musí existovat statická metoda, která ji navrací. Vlastní realizaci tohoto vzoru v aplikaci je v příloze C.1.

Mezi další důležitou funkci, kterou plní pomocná třída ***CGHelper*** je správa, řízení a organizace jednotlivých buněk v mřížce. Tato třída si drží **dvourozměrné pole**, které se bude dále nazývat **maticí**, jehož velikost odpovídá počtu definovaných řádků a sloupců,



Obrázek 9: Realizace návrhového vzoru *Singleton*

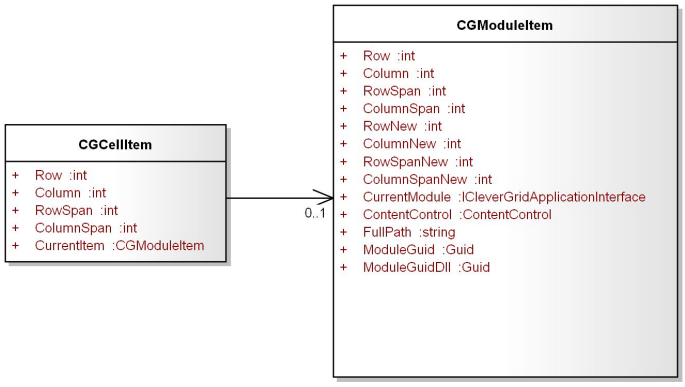


Obrázek 10: Struktura buněk mřížky

které bylo zmíněné v kapitole 4.1. Pro jednotlivé položky je vytvořena speciální struktura a její model je na obrázku 10.

Tento diagram zachycuje, jak je v matici organizována struktura jednotlivých položek mřížky. Vlevo je vidět matice, jejíž struktura odpovídá načtené mřížce. V každé buňce matice je právě jedna instance objektu typu ***CGCellItem***. Jeho přesná struktura bude popsána dále. Dále pak obsahuje objekt ***CGModuleItem***, který představuje modul vložený do mřížky. Každá položka v buňce mřížky může ukazovat na jeden prvek tohoto typu. Tedy pokud je na konkrétní pozici buňky modul vložen. Pak si buňka drží jeho instanci, pokud není, tak je tento ukazatel prázdný. Konkrétní strukturu těchto položek je vidět na následujícím diagramu, který je zobrazen na obrázku 11.

Struktura objektu ***CGCellItem*** je tvořena několika položkami. Proměnná *Row* definuje řádek buňky, *Column* určuje sloupec, *RowSpan* a *ColumnSpan* jsou pomocné proměnné pro další výpočty. Poslední položkou je *CurrentItem*, která je právě typu ***CGModuleItem***. Ta reprezentuje ukazatel na konkrétní instanci tohoto typu, která představuje modul. Celý



Obrázek 11: Diagram položek v mřížce

nebo jeho část se tedy musí nacházet na pozici, která odpovídá této položce. ***Matrix*** neboli **matice** se inicializuje po prvním požadavku na vytvoření instance třídy ***CGHelper***. To je realizováno postupným průchodem všech buněk v matici. Každé se přitom nastaví nová instance ***CGCellItem*** s příslušným řádkem a sloupcem. Odkaz na modul není přitom nastaven.

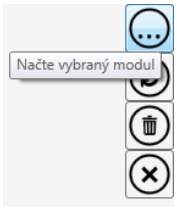
4.2 Realizace modulární architektury

Poté co byla prezentována organizace jednotlivých buněk v **inteligentní mřížce**, je možno se začít věnovat problematice týkající se práce s konkrétním modulem a řešit možná rizika z ní vyplývající. V této části jsou detailně vysvětlovány principy fungování jednotlivých algoritmů. Mezi ně patří:

- Načtení nového modulu do systému nebo jeho vložení do **inteligentní mřížky**.
- Změny velikosti nebo pozice modulů.
- Přemístění nebo změna velikosti ***slave*** modulů.
- Vyhledání optimálních stavů a velikostí ***slave*** modulů.
- Přepínání mezi stavy.
- Implementace nového modulu a nutné podmínky, které modul musí splňovat.

4.2.1 Připojení a odpojení

Jednou ze základních funkcí systému je možnost načtení nového modulu do aplikace. Aby takovýto modul mohl být jeho součástí, musí splňovat důležité podmínky. Díky nim je možno s ním komunikovat a dále jej spravovat. Jak přidat modul je možno vidět na obrázku 12.



Obrázek 12: Načtení nového modulu

4.2.1.1 Nutné podmínky pro připojení modulu

Modulární systém umožňuje za běhu do aplikace dynamicky připojovat nebo odpojovat moduly. Nejjednodušší cestou jak takový systém naimplementovat je možnost využít technologie, která se jmenuje **reflexe**. Detailní informace o ní jsou v příloze D. Díky této technologii se dá docela jednoduše problém dynamického načítání vyřešit. Modul proto musí splňovat jistou množinu podmínek. Na výpisu 8 je algortimus, jak se takový modul v aplikaci načítá.[4]

```
Assembly assembly = Assembly.LoadFile(p_FileName);
foreach (Type item in assembly.GetTypes())
{
  if (item != typeof(UserControl) && !item.IsSubclassOf(typeof(UserControl)))
  {
    continue;
  }
  foreach (Type i in item.GetInterfaces())
  {
    if (i == typeof(ICleverGridApplicationInterface))
    {
      object[] attributes = item.GetCustomAttributes(false);
      if (attributes != null && attributes.Length > 0)
      {
        foreach (object attribute in attributes)
        {
          if (attribute is CleverGridApplicationModuleAttribute && ((
              CleverGridApplicationModuleAttribute)attribute).IsModule)
          {
            using (new WaitCursor())
            {
              ICleverGridApplicationInterface current =
                (ICleverGridApplicationInterface)Activator.CreateInstance(item);

              current.LoadNumberModule(_LastNumberModule);
              _LastNumberModule++;

              if (_GridHelper.LoadInfoFromService)
              {
                AddNewModuleFactory(current, p_FileName, item.GUID);
                return;
              }
              if (!AddNewModule(current, p_FileName, item.GUID))
              {
```

```
                    System.Windows.MessageBox.Show(@"Nebyla nalezena volná pozice pro nový
                        modul, zmenšete některý z již načtených modulů.");
                }
                return;
            }
        }
    }
}
```

Výpis 8: Algoritmus načtení nového modulu

Nyní budou postupně popsány jednotlivé kroky a podmínky, které modul musí splňovat, aby s ním aplikace byla schopna komunikovat.

1. Základní podmínkou je, že každý modul musí být samostatný projekt typu *ClassLibrary*.

2. Načte se příslušná *assembly*. Pomocí **reflexe** se zjití všechy typy, které *assembly* používá a postupně se po jednom prochází.

3. Ověří se, zda aktuální typ je typu *UserControl* nebo z tohoto něj dědí a pokud ne, tak se pokračuje dalším typem.

4. Následně se prochází seznam všech rozhraní, které aktuální typ implementuje. Mezi nimi se hledá rozhraní ***ICleverGridApplicationInterface***. To je možno najít v projektu ***Lib.CleverGridApplicationInterface*** a kapitole 4.6. Toto rozhraní musí implementovat každý modul použitý v aplikaci.

5. Všech implementované moduly dědí ze stejného předka, který implementuje toto rozhraní. Díky tomu rozhraní implementuje i samostatný modul. Je nutné tedy odlišit modul od jeho předka. Proto je nutné přidat ke každému modulu atribut. Tímto atributem je ***CleverGridApplicationModuleAttribute***. Ten je obsažen v projektu ***Lib.CleverGridApplicationLibrary*** popsaném v kapitole 4.6. Pokud objekt obsahuje tento atribut a hodnota jeho proměnné ***IsModule = True***, pak se jedná o objekt, který lze do aplikace připojit.

6. Posledním krokem je vytvoření instance tohoto objektu pomocí metody ***CreateInstance***.

4.2.1.2 Připojení modulu Pokud se podařilo vytvořit tento objekt, je na čase zkusit jej vložit do mřížky. **Testovací** moduly zobrazují číslo, které prezentuje, v jakém pořadí byl daný modul do aplikace vložen. Tuto informaci hlídá proměnná ***_LastNumberModule***, která se s každým nově připojeným modulem inkrementujeme. Její hodnota je do něj načtena a zobrazena. Následujícím krokem je vložení objektu vytvořeného v kapitole 4.2.1.1 do mřížky. Tato část je tvořena několika bloky. Jedním z nich je určení velikosti

modulu a jeho umístění. Princip algoritmu, který tento problém řeší, je popsán v kapitole 4.4.3. Po zjištění těchto informací, je možno vkládat modul. Základem je vytvořit objekt ***CGModuleItem***, který byl na obrázku 11. Na výpisu 9 je vysvětleno, jak se takový objekt vytváří.

```
private void AddNewItemToCurrentPozition(int p_Row, int p_RowSpan, int p_Column, int
    p_ColumnSpan, ICleverGridApplicationInterface p_Module, string p_FullPath, Guid
    p_ModuleGuidDll)
{
  ContentControl cc = new ContentControl()
  {
    VerticalAlignment = System.Windows.VerticalAlignment.Stretch,
    HorizontalAlignment = System.Windows.HorizontalAlignment.Stretch,
  };

  p_Module.MoveClick += new EventHandler(_GridHelper.Module_MoveClick);
  p_Module.ResizeClick += new EventHandler(_GridHelper.Module_ResizeClick);
  p_Module.RemoveModuleClick += new EventHandler(_GridHelper.Module_RemoveModuleClick);

  cc.Width = p_ColumnSpan * _GridHelper.WidthCell - 2.5;
  cc.Height = p_RowSpan * _GridHelper.HeightCell - 2.5;

  Canvas.SetTop(cc, _GridHelper.HeightCell * p_Row);
  Canvas.SetLeft(cc, _GridHelper.WidthCell * p_Column);

  _CurrentSource = new CGModuleItem()
  {
    CurrentModule = p_Module,
    Row = p_Row,
    Column = p_Column,
    ColumnSpan = p_ColumnSpan,
    RowSpan = p_RowSpan,
    FullPath = p_FullPath
  };

  cc.Content = _CurrentSource.CurrentModule as UIElement;

  _CurrentSource.ContentControl = cc;
  _CurrentSource.ModuleGuid = Guid.NewGuid();
  _CurrentSource.ModuleGuidDll = p_ModuleGuidDll;
  _GridHelper.AddControlToGrid(_CurrentSource);
}
```

Výpis 9: Algoritmus vložení nového modulu

1. V první části se vytváří *ContentControl*, který bude umístěn v mřížce. Zaobaluje modul a díky němu je možno manipulovat s modulem na *Canvasu*.
2. Registrují se události pro manipulaci s modulem.
3. Nastavují se příslušné velikosti *ContentControlu*, které jsou odvozené od velikosti buňky mřížky a počtu buněk, které modul zabírá.

4. Dále je pak nutné, nastavit jeho pozici v *Canvasu*.

5. Nyní je možno začít vytvářet *CGModuleItem*, který prezentuje modul. Nastavují se informace týkající se pozice a velikosti modulu. Jelikož některé moduly si potřebují ukládat dočasné informace, předává se i umístění aplikace v adresářové struktuře. V ní si jednotlivé moduly vytvoří svoji složku a tato dočasná data v ní zde udržují. Poté se předává instance rozhraní ***ICleverGridApplicationInterface***.

6. Vytvořený objekt ***CGModuleItem*** se vloží do obsahu *ContentControlu*. Ten se zase přidá do objektu ***CGModuleItem*** a vytvoří se oboustranná vazba.

7. Posledním krokem při vytváření této položky je nastavení dvou identifikátorů, které slouží k odlišení jednotlivých typů a instancí modulů. To slouží pro práci se sociálním chováním. Ta je vysvětlena v kapitole 5.1.2.

Nyní je vytvořen objekt ***CGModuleItem*** a je možno pokračovat ve vkládání do mřížky. Tato část se skládá ze tří kroků.

1. Vložení *ContentControlu* do komponenty ***Grid***. Následně je nutné této komponentě předat informace o jeho velikosti a pozici. Jedná se tedy o nastavení příslušného řádku a sloupce pro konkrétní pozici a počtu sloupců a řádku pro velikost, kterou budou v mřížce zabírat.

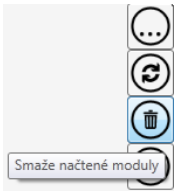
2. Obsazení příslušných volných buněk v **matici**. Procházejí se tedy postupně jednotlivé ***CGCellItem*** buňky na konkrétních pozicích a jim se pak nastavuje odkaz na tuto konkrétní instanci objektu ***CGModuleItem***. Pokud je nastaven tento odkaz, buňka se pak jeví jako obsazená.

3. Přidání instance ***CGModuleItem*** do **kolekce všech modulů**.

4.2.1.3 Odpojení modulu Kromě připojení, umožňuje aplikace taky odpojit konkrétní modul. Tuto funkci provádí tlačítko se symbolem písmena X, které obsahuje v hlavičce každým implementovaný modul v systému. Realizace funkce odstranění modulu je velmi jednoduchá a algoritmus, který ji řeší, je v příloze C.2. Registrace události, která vyvolá tuto metodu, je na výpisu 9. Vyvolání, předání a typy parametrů události jsou vysvětleny v kapitole 4.3. Realizace odpojení je tvořena několika kroky.

1. Je nutno zjistit, o jaký modul se jedná. To se zjistí z parametru, který vyvolaná událost předává. Ten je typu *UserControl*, který představuje aktuální modul. Díky vlastnoti *Parent* tohoto objektu je možno zjistit, o který *ContentControl* se jedná. To je možno si dovolit, protože jak již bylo zmíněno v kapitole 4.2.1.2, do obsahu *ContentControlu* se vkládá modul. Tedy objekt, který dědí z *UserControlu*.

2. Následuje vyhledání umístění modulu a jeho velikost. Poté je možno jej odebrat z mřížky.



Obrázek 13: Smazání všech modulů

3. Uvolnění pozic, které zabíral. To se realizuje pomocí získaných informací o pozici a velikosti, kdy se opět prochází jednotlivé buňky ***CGCellItem*** z **matice** a postupně se jim ruší odkaz na modul, tedy ***CGModuleItem***, na který ukazovaly. Tím se nastaví buňce příznak, že je již volná.

4. V poslední fázi se odebere modul z kolekce připojených modulů.

Dále aplikace nabízí ještě jednu obdobnou funkci. Tou je hromadné odstranění všech modulů, které byly do systému připojeny. Tuto funkci je možno vyvolat kliknutím na příslušné tlačítko, které se nachází v pravé liště nástrojů aplikace. Toto tlačítko je vidět na obrázku 13.

Tato funkce je obdobná, jako mazání jednotlivých modulů. Zde se opět nejprve odeberou všechny moduly z mřížky. Poté se všem buňkám, které ukazovaly na nějaký modul, odebere vazba na konkrétní modul. V posledním kroku se vynuluje kolekce všech načtených modulů.

4.2.1.4 Automatické načítání a ukládání použité konfigurace

V průběhu implementace bylo nutno neustále zapínat a vypínat aplikaci. Po spuštění bylo potřeba mnohokrát znovu načítat stejné moduly a umisťovat je na pozice, na které byly při předchozím spuštění vloženy. Toto bylo velmi pracné, a proto je implementována tato pomocná funkce, které automaticky po vypnutí aplikace ukládá tyto informace a po jejím spuštění je schopna je opět načíst a dané moduly nakonfigurovat.

Tyto funkce byly občas nežádoucí. Proto je umožněno, aby si uživatel mohl libovolně konfigurovat, kterou z daných možností chce mít zapnutou a kterou naopak ne. To je možno nastavovat v konfiguračním souboru celé aplikace. Pro ukládání je určen klíč *SaveModuls* a pro načítání klíč *LoadModuls*. Jednotlivé konfigurace se ukládají do pomocného souboru ve formátu *XML*. Formát toho souboru po zavření aplikace a nastaveném parametru *SaveModuls* na *True* je zobrazeno na výpisu 10.

```
<?xml version="1.0" encoding="UTF−8"?>
<Configuration>
  <Moduls>
    <Module FullPath="D:\Gray.dll" Row="4" Column="7" RowSpan="3" ColumnSpan="3" />
    <Module FullPath="D:\Blue.dll" Row="5" Column="4" RowSpan="3" ColumnSpan="3" />
    <Module FullPath="D:\Blue.dll" Row="2" Column="4" RowSpan="3" ColumnSpan="3" />
    <Module FullPath="D:\Red.dll" Row="1" Column="7" RowSpan="3" ColumnSpan="3" />
    <Module FullPath="D:\Orange.dll" Row="2" Column="10" RowSpan="3" ColumnSpan="3" />
    <Module FullPath="D:\Blue.dll" Row="5" Column="10" RowSpan="3" ColumnSpan="3" />
```

```
    </Moduls>
</Configuration>
```

Výpis 10: Struktura souboru pro automatické načtení a uložení konfigurace mřížky

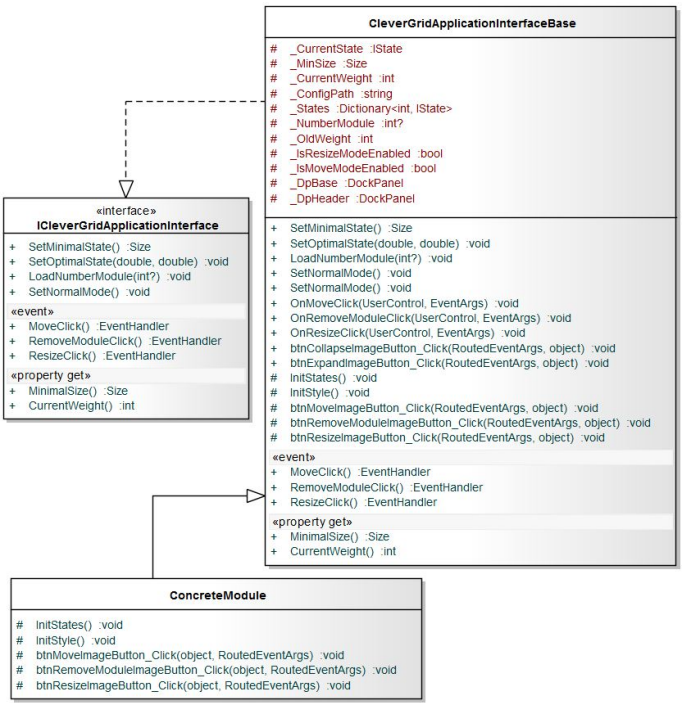
Pro následující konfiguraci bylo před zavřením aplikace načteno šest modulů. Jednotlivé cesty jsou uloženy v parametru *FullPath*. Konfigurace umístění a původní velikosti modulu jsou uloženy ve zbylých čtyřech atributech. Pro ukládání těchto informací je použit jednoduchý algoritmus. Ten spočívá ve vytvoření dočasného *XML* souboru. V něm je vytvořen kořenový element *Moduls*. Poté se projde postupně **kolekce připojených modulů**, ve které jsou uloženy všechny moduly prostřednictví objektu *CGModuleItem*. Ten v sobě obsahuje všechny potřebné informace. Jednotlivé názvy proměnných v tomto objektu si odpovídají s názvy atributů v *XML* souboru. Pro každý záznam v kolekci je vytvořen jeden element *Module* s příslušnými hodnotami jednotlivých atributů. Ten je pak přidán do kořenového elementu *Moduls*.

Načtení této konfigurace po zapnutí aplikace je obdobně jednoduché. Pokud existují, tak se postupně prochází jednotlivé elementy *Module* z kořene *Moduls*. Pokud obsahuje všechny potřebné informace, tak se využije stejného algoritmu jako v kapitole 4.2.1.1. Pokud v konfiguraci souboru nastane situace, že pro modul je určitá buňka již obsazena, tak se tento modul do systému nenačítá a pokračuje se dalším elementem.

4.3 Implementace modulu

Vytvoření nového modulu, který lze připojit do aplikace a dále s ním pracovat je velmi jednoduché. Pro snadné pochopení problematiky je zde prezentována jeho implementace na již vytvořeném modulu. Bude se jednat o **testovací** modul ***Blue***. Tento modul nemá žádnou vedlejší funkcionalitu a byl vytvořen pouze pro testovací účely. Díky tomu budou vidět pouze funkce, které skutečně souvisí pouze s jeho implementací.

Modul je vlastně formulář s uživatelským rozhraním. Jelikož je systém implementován technologií *WPF*, tak se jedná o objekt, který nutně musí dědit z prvku *UserControl*. Dále pak musí implementovat rozhraní *ICleverGridApplicationInterface*, které poskytuje funkce pro komunikaci mezi modulem a aplikací. Všechny důležité objekty tohoto rozhraní je možno vidět na diagramu, který je zobrazen na obrázku 14.



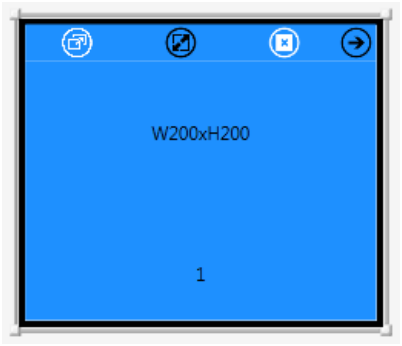
Obrázek 14: Struktura implementace a dědičnost modulu

Jelikož při implementaci jednotlivých modulů byly nalezeny situace, kdy se velké množství stejných zdrojových kódů velmi často opakovalo na různých místech, bylo využito **dědičnosti**. Proto byl navržen společný předek pro všechny implementované moduly, který se jmenuje ***CleverGridApplicationInterfaceBase*** a dále se bude nazývat **předkem**. Je umístěn v projektu ***Lib.CleverGridApplicationWpfLibrary***. Jeho smysl je vysvětlen v kapitole 4.6. Jednotlivé moduly z něj dědí. To hodně usnadnilo práci nejenom tím, že nebylo nutné stejné kódy stále kopírovat, ale pokud byla provedena nějaká změna v **předkovi**, projevila se snadno u všech modulů. Při jeho implementaci byl však nalezen jeden zásadní problém. Jednalo se o to, že technologie *WPF* nepodporuje zcela ideálně dědičnost mezi *UserControly*. To znamená, že *XAML* formulář nemůže dědit přímo z *XAML* formuláře. Tuto dědičnost lze realizovat pouze tak, že formulář může dědit pouze z objektu, který je třída. Ta může dědit z *UserControlu*. Z toho plyne, že veškeré grafické prvky, které byly prezentovány v tomto předkovi, bylo nutné složitě definovat pomocí jednotlivých objektů a jejich vlastností v této třídě.

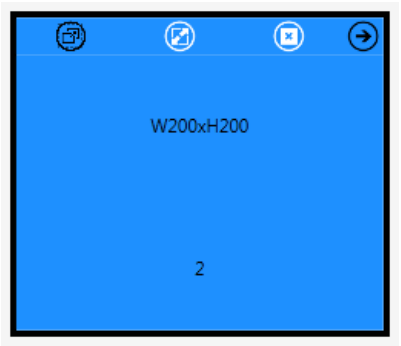
Dalším důležitým aspektem, který vyplývá z použití dědičnosti je jednotný vzhled všech modulů. Tím je myšleno grafické rozložení jednotlivých důležitých prvků. Mezi ně patří jak umístění vlastního grafického obsahu modulu, tak i především ovládání chování formuláře. Tedy nástroje pro odstranění, změnu velikosti nebo pozice modulu.

Předek je tvořen základní komponentou *DockPanel*, která je schopna svůj obsah jednoduše roztáhnout přes celý formulář. Na obrázku 15 je vidět, že je tento panel tvořen dvěma základní bloky.

Obrázek 15: Základní bloky modulu



Obrázek 16: Mód pro změnu velikosti modulu



Obrázek 17: Mód pro přesun modulu

První z nich obsahuje tlačítka pro práci s formulářem. Jedná se o první tři bílá tlačítka zleva na obrázku 15. Poslední tlačítko slouží pouze pro zobrazování nebo skrývání tohoto panelu. Pokud se tlačítko zbarví do černa, je aktivován příslušný mód. Registrace událostí pro přepnutí do konkrétního módu nebo k odstranění modulu je tvořena při vkládání nového modulu a je vidět na výpisu 9. Předek si registruje jednotlivé události těchto tří tlačítek. Metody, které jsou zavolány pro odchycení konkrétní události, jsou *virtuální* s modifikátorem přístupu *protected*, proto aby s nimi mohl konkrétní potomek, tedy modul dále nakládat. Uvnitř se provádí několik operací, které jsou shodné pro všechny moduly. Mezi ně patří například změna barvy tlačítka. Pokud se jedná o událost nastavení stylu pro možnost změny pozice, je nastavena příslušná hodnota vlastnosti *IsHitTestVisible* objektu *_DpBase*. Ta určuje, zda bude přístupný kurzor pro změnu pozice tohoto objektu na komponentě *Canvas*. Dále je pak provedena podstatná kontrola, že daný modul může být pouze v jednom ze tří možných módů. Jedná se tyto módy:

- **Normální mód** - obrázek 15.
- **Mód pro změnu velikosti** - obrázek 16.
- **Mód pro změnu pozice** - obrázek 17.

Po provedení těchto operací si tuto metodu dále zpracovává potomek. Možné přepsání jednotlivých událostí a předání příslušných parametrů je zobrazeno na výpisu 11.

```
protected override void btnResizeImageButton_Click(object sender, RoutedEventArgs e)
{
  base.btnResizeImageButton_Click(sender, e);
  OnResizeClick(this, new EventArgsBase(_IsResizeModeEnabled, null));
}

protected override void btnMoveImageButton_Click(object sender, RoutedEventArgs e)
{
  base.btnMoveImageButton_Click(sender, e);
  OnMoveClick(this, new EventArgsBase(null, _IsMoveModeEnabled));
}
```

```
protected override void btnRemoveModuleImageButton_Click(object sender, RoutedEventArgs e)
{
  OnRemoveModuleClick(this, new EventArgs());
}
```

Výpis 11: Zpracování událostí pro manipulaci s modulem

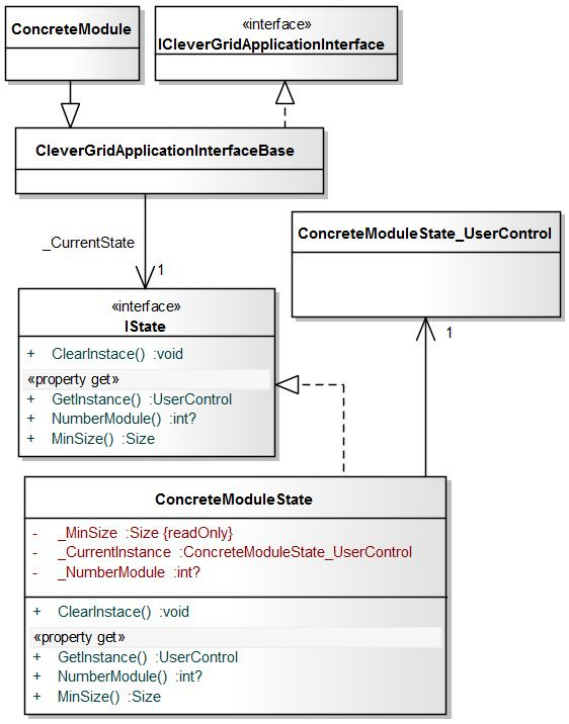
- Při vyvolání přepnutí do konkrétního módu nebo mazání, se volá příslušná **přepsaná** metoda události modulu.
- Poté se provedou kontoly a funkce v předkovi.
- Do objektu sender se zasílá konkrétní instance modulu. Díky tomu je systém schopen při odchycení události v aplikaci zjistit, o jaký objekt se jedná. Toto odchycení bylo již použito při mazání modulu v kapitole 4.2.1.3.
- Pro přenos argumentů události se používá vlastní objekt *EventArgsBase*, který je potomkem *EventArgs*. Tato třída je z projektu *Lib.CleverGridApplicationLibrary*, který je částí obsahu kapitoly 4.6. Do toho objetu se zasílá informace, zda jde o aktivaci či deaktivaci konkrétního módu.

  Algoritmus pro nastavení stylu změny velikosti modulu je zobrazen v příloze C.3.
- Pokračuje se v ověření, zda se bude nastavovat styl nebo je již nastaven. Pak se tedy jedná o přepnutí do normálního módu a styl se modulu musí odstranit.
- Systém podporuje možnost pouze jednoho modulu, který je v jiném módu než v normálním. Pokud se tedy konkrétní modul přepíná do módu pro změnu velikosti nebo pozice, je nutné všem ostatním modulům nastavit normální mód. Musí se jim odstranit styl. To je realizováno tak, že se projde kolekce modulů a všem kromě aktuálního se styl zruší. Definování a význam těchto stylů je popsán v kapitole 4.1.1.

Druhým blokem tohoto předka je vlastní grafická a funkční prezentace modulu. Pro jeho úplné roztažení velikosti je tvořen opět komponentou *DockPanel* jejíž konkrétní instance se jmenuje *_DpBase*. Zde se realizuje většina metod z implementovaného rozhraní. Tyto metody mají většinou sufix *State*. Jedná se tedy o práci se stavy.

Vzhled **testovacích** modulů v závislosti na jejich velikosti je až na malé detaily stále stejný. Problém nastává při práci s **funkčními** moduly. Například modul **RssModule**, který prezentuje informace z *RSS* kanálů. Pokud bude jeho velikost nastavena na 10x10 pixelů, tak z něj zřejmě žádné důležité informace nebudou čitelné. Proto mají jednotlivé moduly v závislosti na jejich velikosti různé stavy. Ty jsou těmto velikostem přizpůsobeny a při její změně se automaticky přepínají. Díky tomu je možno pak tyto informace předávat a prezentovat v čitelné podobě. Práci s těmito stavy předek plně podporuje. Tato situace přímo vybízí k použití návrhového vzoru. Tím je vzor **State**, který je popsán v příloze B.2.

Implementace a práce se stavy je inspirována tímto vzorem a je snaha ho do aplikace co nejlépe zakomponovat. Každý takovýto stav musí implementovat rozhraní ***IState*** z



Obrázek 18: Struktura pro práci se stavy modulu

projektu *Lib.CleverGridApplicationWpfLibrary* z kapitoly 4.6. Struktura, která byla navržena pro práci se stavy, je zobrazena na diagramu z obrázku 18.

Objektem, který toto rozhraní implementuje, je vždy nějaká třída. Té vždy odpovídá jeden příslušný *UserControl*. Každý stav má definovanou **minimální velikost**. Je to z důvodu odlišení jednotlivých stavů od sebe. Ta se nastavuje v závislosti na informacích, které se v konkrétním stavu zobrazují. Každá z těchto tříd dále obsahuje odkaz na *UserControl*, který danému stavu přísluší. Ten je realizován vlastností *GetInstance*, která vytváří příslušný objekt *_CurrentInstance*. Prostřednictvím ní je tato instance přístupná přes rozhraní. Posledním důležitým objektem diagramu z obrázku 18 je proměnná *NumberModule* obsahující pořadí, ve kterém daným modul byl vložen do mřížky.

Každý modul, který dědí z předka, si při vytváření inicializuje kolekci všech jeho dostupných stavů. O to se stará metoda *InitStates*, která je v předkovi *virtuální*. Každý modul by si ji měl přepisovat. Tato kolekce je tříděna pomocí **vah**. Každý stav je ohodnocen podle své velikosti. Aktuální stav je uložen v proměnné *CurrentState* a jeho váha v *CurrentWeight*. Při vytvoření modulu je vždy tento odkaz nastaven na takzvaný **minimální stav**. To je stav s nejmenší váhou. Minimální velikost tohoto stavu je nastavena do proměnné *MinSize*. Díky tomu je pak možno zjistit, že pokud dojde k pokusu o zmenšení modulu a jeho velikost je menší než tato proměnná, tak modul nelze na takovou velikost zmenšit.

Při vytváření objektu aktuálního stavu se nastavuje vzhled modulu, tedy jeho styl. To se provádí přepsáním metody *InitStyle*. O jednotlivé přepínání mezi stavy se stará metoda předka *SetOptimalState*. Ta na základě aktuální šířky a výšky modulu rozhoduje, zda je možno vůbec na takovou velikost modul nastavit. Následuje hledání optimálního stavu, které je realizováno postupným procházením kolekce stavů. Mezi nimi se hledá stav s nejvyšší váhou. Jeho minimální šířka i výška musí být menší než hodnoty aktuální velikosti modulu. Poté se zkontroluje, jestli váha nalezeného stavu je odlišná od té původní. Pokud ano, je původní stav odstraněn a vytvoří nová příslušná instance aktuálního stavu. Nakonec se nastaví pořadí, ve kterém byl modul do aplikace vložen.[5, 6]

4.4 Manipulace s modulem

V předchozích kapitolách byly prezentovány algoritmy, které realizují základní funkce systému. Nyní bude soustředěna pozornost na algoritmy, které řídí chování jednotlivých modulů. Některé z těchto funkcí byly již lehce zmíněny v některých z předešlých kapitol. Mezi tyto algoritmy patří:

- Určení nové pozice či velikosti ***master*** modulu.
- Vyhledávání volných míst v mřížce.
- Vyhledání nové pozice či velikosti pro ***slave*** modul, který byl ovlivněn v důsledku přesunu nebo změny velikosti ***master*** modulu.
- Určení startovací pozice pro vyhledávání.
- Určení nejvhodnější volné plochy.
- Nastavení nejvhodnějšího stavu konkrétního ***slave*** modulu.
- Optimalizace velikosti modulu v závislosti na poměru stran plochy a konkrétním stavu.

4.4.1 Přesun modulu

První z důležitých algoritmů, které řídí chování modulů, je funkce pro změnu pozice modulu. Pro její změnu se využívá objektu *MoveThumb* z kapitoly 4.1, který obsahuje registraci a odchycení důležité události *DragDelta*. Po jejím odchycení dochází ke zjištění, zda je možno modul přesunout.

V první řadě je nutno zjistit o jaký objekt jde. Prostřednictvím stylu z kapitoly 4.1.1 je možno díky *DataContextu* toho objektu zjistit, o jaký modul se jedná. Následuje zjištění původního umístění prvku v mřížce. Jako další se ověřuje, zda opravdu došlo ke změně pozice. To se nejprve provádí pro řádky a následně pak pro sloupce. Pokud došlo k posunu, nastavuje se nová pozice. Posledním krokem je zavolání **funkce pro zpracování a ověření nové pozice**. Pokud však pozice nebyla změněna ani pro jeden ze směrů, tak algoritmus končí. Tento algoritmus je zobrazen v příloze C.4.

Hlavní úlohou **funkce pro zpracování a ověření nové pozice *master*** modulu je zjistit, zda je možno daný ***master*** modul vůbec přemístit. To se nemusí podařit, pokud jeho nové umístění společně s jeho velikostí překrývá pozice, kde byl jeden nebo více jiných modulů. Tyto ***slave*** moduly se pak pokouší přemístit nebo měnit jejich velikosti. Zdrojový kód algoritmu je v příloze C.5.

1. V první řadě je nutno si vytvořit **klon** původního rozpoložení objektů v **matici**. Je to proto, že se dále provádí kontroly, zda je možno vůbec ***master*** modul někam přesunout a díky tomu je zachována původní konfigurace.

2. Následuje vyhledání všech ***slave*** modulů, které se budou muset přemístit nebo zmenšit. To je realizováno postupným průchodem všech buněk **matice**. Interval, ve kterém se hledá, je omezen pozicí modulu a jeho velikostí. Všechny prvky, které sem alespoň částečně spadají, jsou označeny jako ***slave***. Původním buňkám těchto modulů, které jsou překryty, je nastaven odkaz na ***master*** modul.

3. Poté se postupně zpracovávají jednotlivé ***slave*** moduly. Pro každý z nich se provede posloupnost několika operací.

4. Nejprve se provede jeho odebrání z **klonu matice**. Tím se uvolní místo a je možno hledat jeho nové umístění nebo změnu velikosti.

5. Následuje určení pozice, od které se začne hledat. To je popsáno v kapitole 4.4.4. Pokud se nepodaří tuto pozici najít, znamená to, že není v matici již volné místo. Pak se nastavuje příznak, že některý z ***slave*** modulů není možno přemístit. Algoritmus je ukončen a původní konfigurace modulů se nepřepisuje.

6. Pokud je však pozice nalezena, vyzkouší se, zda není možno tento modul vložit v původní velikosti přímo na tuto nalezenou pozici. Jestli to možno není, hledá se spirálově nejbližší volná plocha odpovídající velikosti aktuálně zpracovávanému modulu. Vyhledávání je vysvětleno v kapitole 4.4.3.

7. Jestli se však nepodařila najít, je nutné zkusit jeho velikost zmenšit. Je použito algoritmu pro **nalezení nejbližší menší volné plochy**, na kterou by se tento modul po zmenšení mohl vejít. Jak je možno tyto plochy hledat je vysvětleno v kapitole 4.4.3.

8. Jelikož tato funkce může najít více ploch, je nutné mezi nimi najít tu nejoptimálnější. Optimalizace je popsána v kapitole 4.4.5 a informace o stavech v kapitole 4.3.

9. Následuje ověření, zda byla takováto plocha nalezena. Pokud ne, tak algoritmus končí a ***master*** modul nelze přesunout. Jestli se ji však podařilo najít, je ***slave*** modulu nastavena nová velikost a pozice.

10. Následně se ještě provede ověření, zda není možno výsledné umístění aktuálně zpracovávaného modulu posunout co nejblíže ***master*** modulu.

11. Jako poslední společný krok pro ***slave*** moduly je nastavení příslušné velikosti a pozice do **klonu matice**.

12. Dalším blokem je vyhodnocení příznaku, zda se pro všechny ***slave*** moduly podařilo najít nové volné pozice či velikosti.

13. Pokud nebyly nalezeny žádné ***slave*** moduly, pak se nastavuje pouze pozice a velikost ***master*** modulu do **matice** a jeho optimální stav pomocí algoritmu z kapitoly 4.3.

14. Pokud bylo zjištěno, že není možno ***master*** modulu změnit pozici, nastaví se původní konfigurace.

15. Poslední možností je, že byly nalezeny ***slave*** moduly a podařilo se je přemístit či změnit jejich velikost. Pak je matice přesána jejím klonem. Původní konfigurace matice je přepsána aktuálním stavem nových umístění jednotlivých modulů. Dále se pak jednotlivým ***slave*** modulům nastavuje nová pozice a velikost do mřížky. Přidá se informace o sociálních prvcích, která je popsána v kapitole 5.1.2. V posledním řadě se toto nastaví i ***master*** modulu.

4.4.2 Změna velikosti modulu

Mezi další možnost, jak pracovat s modulem, je funkce pro **změnu velikosti modulu**. Ta je realizována obdobně, jako je **změna pozice**. Zde se využívá třídy *ResizeThumb* a opět události *DragDelta*. Její odchycení a zpracování je velmi podobné, jako bylo u **změny pozice** v kapitole 4.4.1. Realizace toho algoritmu je rozdělena do dvou částí.

V první z nich se zjišťuje, jakým směrem došlo ke změně velikosti. Tedy **vertikálně** nebo **horizontálně**. V závislosti na tom se pak ověřuje, zda došlo k posunu **doprava** či **doleva**, pro **horizontální** změnu. Posun **nahoru** či **dolů** pro změnu **vertikální**. Tato funkce je popsána v příloze C.6.

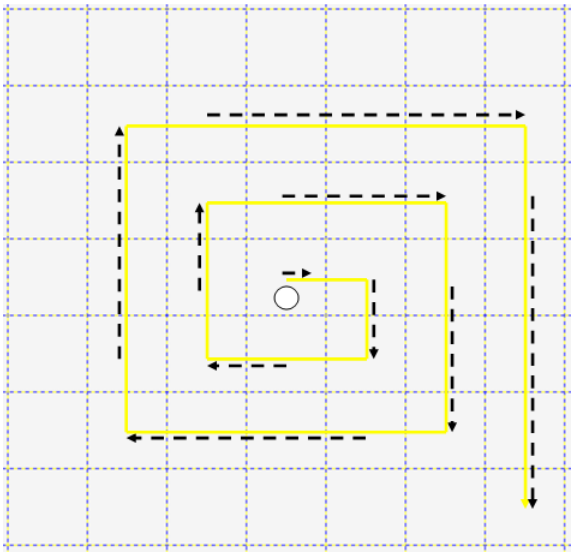
Jelikož je princip pro všechny čtyři směry obdobný, byl vybrán blok, který řeší změnu velikosti v **horizontálním** směru. V první řádě je nutno zjistit, jestli byl použit **pravý** nebo **levý** posuvník. Pro tento příklad byl vybrán **levý**. Dále je pak důležité prověřit, jakým směrem bylo tímto posuvníkem taženo.

- Pokud byl posun **vlevo**, tak došlo ke **zvětšení** modulu. Je nutné **zmenšit** hodnotu sloupce a **zvětšit** počet sloupců, které modul překrývá.

- Pokud byl posun **vpravo**, došlo ke **zmenšení** modulu a je nutné udělat opak. **Zmenšit** počet sloupců, které modul překrýval a **zvětšit** hodnotu jeho pozice, tedy sloupce.

Jelikož je aplikace postavena na okamžitém reagování na tyto změny, bude tato hodnota posunu vždy nula nebo jedna.

Poté se ještě kontroluje, zdali daná pozice není mimo mřížku a nastaví se příslušné velikosti modulu. Dále jeho umístění na *Canvasu* a příznaku, že došlo ke změně. To je stejné jako u **změny pozice** v kapitole 4.4.1.



Obrázek 19: Základní směr spirálového vyhledávání

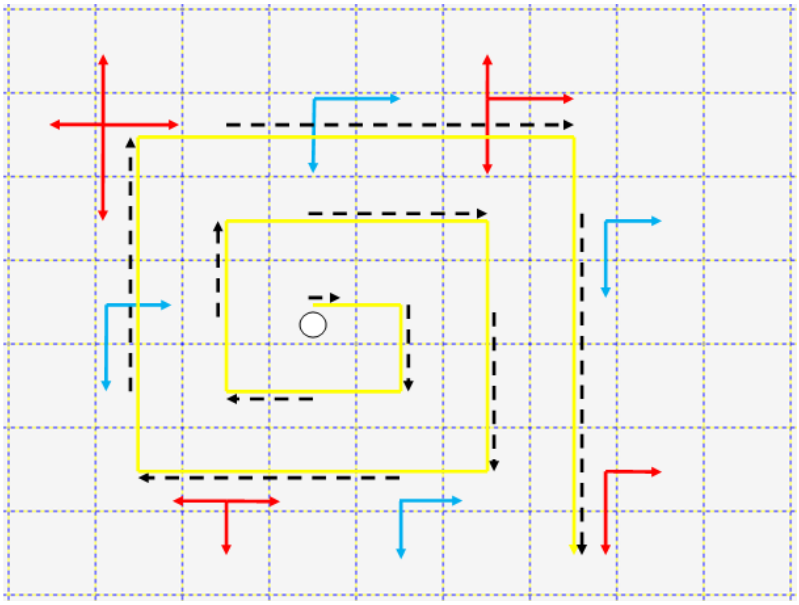
Následuje druhá část, která se zabývá **zpracováním a ověřením nové velikosti *master*** modulu. Také i ověřením, jestli je možno jeho velikost vůbec změnit. Jedná se o obdobu algoritmu, který je popsán v kapitole 4.4.1. Jediným rozdílem je, že se na začátku provede jedna **důležitá kontrola**. Smyslem tohoto ověření je, že se porovnává aktuální nová šířka a výška s tou minimální, která odpovídá ***master*** modulu, jenž byl zvětšen nebo zmenšen. Pokud tato kontrola neproběhne v pořádku a tedy alespoň jedna z aktuálních hodnot šířky nebo výšky je menší než ta minimální, algoritmus končí a ***master*** modulu jsou nastaveny původní hodnoty. Pokud však vše proběhne v pořádku, pokračuje se úplně stejným algoritmem pro **zpracování nové pozice a velikosti *master*** modulu z kapitoly 4.4.1.

4.4.3 Vyhledání volných ploch na základě startovací pozice

Této funkcionality se využívá především při **vkládání nového modulu** do aplikace. Tento modul se do systému vkládá v **minimálním stavu**. Po nastavení tohoto stavu se na základě jeho velikosti zjistí počet řádku a sloupců, které pokrývá. Dalším potřebným vstupem je **startovací pozice**, jejíž zjištění je popsáno v kapitole 4.4.4. Jelikož jsou známy všechny podstatné údaje, tedy **plocha tvořená sloupci a řádky**, dále pak **startovací pozice**, je možno začít popisovat algoritmus. Ten je založen na takzvaném **spirálovém vyhledávání**. Jeho princip je velmi jednoduchý a je ho možno vidět na obrázku 19.

Od **startovací pozice** se posouvá po kruhu postupně **vpravo**, poté hned **dolů**. Jak se narazí na hranu, pokračuje **vlevo** a pak **nahoru**. To se opakuje stále dokola, pokud je alespoň jednou stranou uvnitř mřížky.

Na obrázku 19 je zachycena základní myšlenka tohoto algoritmu. **Bílým kolečkem** je znázorněna **startovací pozice**, od které se bude **spirálově vyhledávat**. **Žlutá šipka** pak



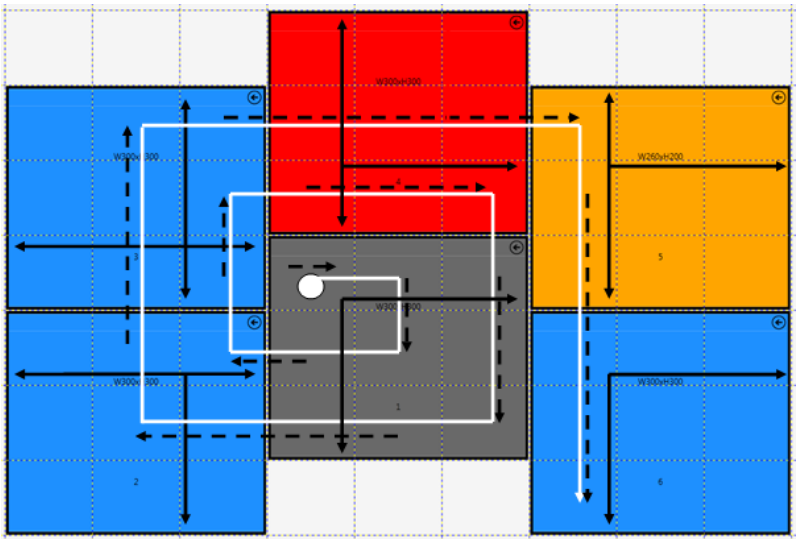
Obrázek 20: Princip algoritmu spirálového vyhledávání pro nalezení konkrétní plochy

zobrazuje vlastní pohyb po spirále a **černé čárkované šipky** znázorňují jednotlivé směry pohybu po spirále.

Jelikož jsou k dispozici čtyři různé směry, je použito čtyř mírně se lišících algoritmů. Každý z nich je však založen na základní myšlence, kdy se od určité pozice hledá plocha vždy ve směru **doprava** a **dolů**. To je postačující pouze u vyhledávání ve směru spirály **dolů**.

V rámci optimalizace jsou zbylé tři strany obohaceny o přidání dalších směrů vyhledávání v rámci jednoho kroku. To má jednoduché vysvětlení. Pokud tedy bude vyhledáván například **směr spirály vlevo**, má dále smysl hledat plochu kromě základních směrů **vpravo** a **dolů** i do **levé** strany. Pokud tedy bude posloupnost buněk **vlevo** od aktuální pozice volná a jejich počet bude odpovídat počtu buněk pro aktuální modul, je možno ho na tuto pozici **vlevo** vložit. Díky tomu dojde k ušetření dalších průchodů **vlevo**, a tak i k optimalizaci algoritmu. Obdobně je tomu i pro směr **vpravo**, kde se opět hledá **doprava** a **dolů**. Pokud se plocha nenajde, hledá se i **nahoru**. Posledním směrem je průchod **nahoru**. Kde se provádí kombinace předešlých dvou směrů. Opět tedy prohledání **vpravo** a **dolů**. Následně se přidají směry **doleva** a **nahoru**. Konkrétní příklady vyhledávání v různých směrech jsou zobrazeny na obrázku 20. **Standardní strany** pro jednotlivé směry vyhledávání jsou zobrazeny **dvěma modrými šipkami**. **Alternativní možné strany** pro vyhledávání jsou naznačeny **červenými šipkami** a to **třemi** pro směr **vlevo** či **vpravo**. **Čtyřmi** pak pro směr **nahoru**.

Na obrázku 21 je zobrazen konkrétní příklad **spirálového vyhledávání** i s moduly. Tady bylo využito algoritmu **náhodného načtení** šesti modulů. Jeho princip je vysvětlen v příloze A.1. **Bílé kolečko** prezentuje **startovací bod**. **Bílou čárou** je zobrazen **spirálový**



Obrázek 21: Příklad načtení modulů spirálovým vyhledáváním

průchod a **černými čárkovanými** pak jednotlivé **směry** tohoto průchodu. **Černými čárami** jsou zobrazeny strany pro hledání **volné plochy** konkrétního modulu.

Prvním modulem, který se podařilo načíst, je ten **šedý**. Algoritmus vždy začíná vyhledávat od **startovací pozice**, pro kterou byl určen směr **vpravo**. Provede se tedy nejprve kontrola **vpravo** a **dolů**, která se dále bude nazývat **standardní**. Je-li vyhodnoceno, že tato plocha je volná, není nutno kontrolovat dále **alternativní** strany a **šedý** modul je na tuto pozici vložen.

Následuje **modrý** modul s číslem **dvě**. První volná **startovací pozice** byla nalezena až pro směr **vlevo**. Opět se provádí **standardní** kontrola. Ta však nevyhovuje. Je proto nutno vyzkoušet **alternativní** a to pro strany **vlevo** a **dolů**. Požadovaná plocha je nalezena o dvě buňky vlevo. Modul je tedy vložen a je možno pokračovat dále.

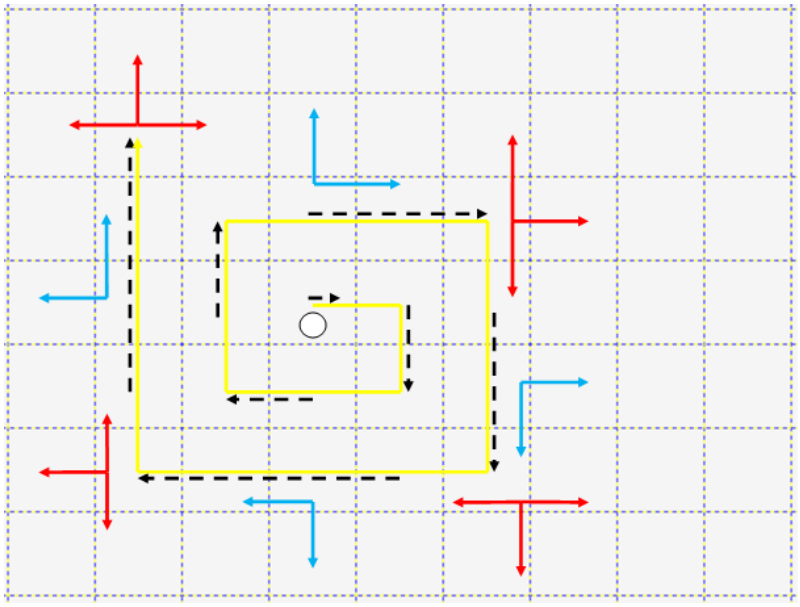
Třetím modulem je opět **modrý**. U něj se podařilo najít volnou **startovací pozici** hned na následující buňce, kdy pro směr **nahoru** nepostačila **standardní** kontrola a bylo nutno zkusit plochu vyhledat **alternativní** možností **vlevo** a **nahoru**.

Dalším modulem je číslo **čtyři** s **červenou** barvou, kde se vyhledává ve směru spirály **doprava**. Opět nepostačila **standardní** kontrola a přidá se směr hledání **nahoru**.

Předposledním modulem je **oranžový** s číslem **pět**, kde se prochází směrem **dolů** a použilo se **alternativního** hledání pro strany **vpravo**, **dolů** a **nahoru**.

Posledním modulem je **modrý** s číslem **šest**, který je také ve směru **dolů**. Nyní však postačila pro nalezení plochy **standardní** kontrola.

Další důležitou funkcí, která využívá **spirálového vyhledávání** a je postavena na obdobě tohoto algoritmu, je hledání **alternativních volných ploch**. To se využívá tehdy, pokud pro ***slave*** modul po změně velikosti nebo pozice ***master*** modulu, nebyla nalezena volná plocha pro jeho původní velikost ***slave*** modulu. Je tedy nutné tento ***slave*** modul zmenšit. Vyhledávání pomocí tohoto algoritmu je zobrazeno na obrázku 22, kde jsou vidět



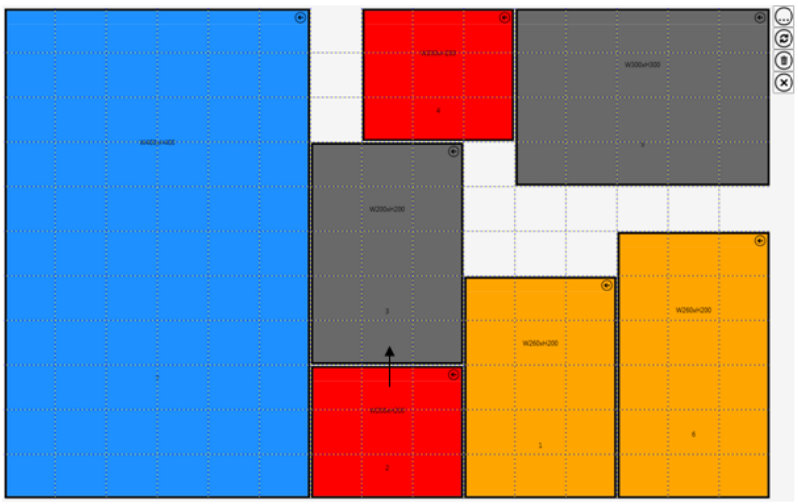
Obrázek 22: Princip algoritmu spirálového vyhledávání volných ploch

různé průchody algoritmu. **Kolečko** prezentuje opět **startovací pozici**. Jednotlivé **černé čárkované čáry** zobrazují postupný průchod po **spirále** od **startovací pozice** a **žlutá** barva prezentuje vlastní **spirálu**. Dále pak obsahuje **červené šipky**, které vycházejí z buňky. Ta prezentuje **rohový bod**. V této buňce se hledá v závislosti na aktuálním směru spirály ve **třech směrech**. **Modré šipky** znázorňují **běžný bod** na **spirále** a postačují hledat pouze ve **dvou** směrech.

Algoritmus od **startovací pozice** po **spirále** hledá volnou buňku. V závislosti na její pozici se různě hledají volné plochy, které se dále zpracovávají. Pokud pro aktuální buňku nebyly nalezeny žádné volné plochy, pokračuje se po spirále další nejbližší volnou buňkou. Toto se opakuje, dokud takové místo není nalezeno.

Pokud se nějakou plochu podaří najít, dále se **zpracovává** a **optimalizuje**. To je popsáno v kapitole 4.4.5. Pokud po zpracování žádná plocha není vhodná, pokračuje se v hledání ploch pro další volnou buňku na spirále. Toto se opakuje, dokud takové místo není nalezeno.

Vyhledávání takovýchto ploch se provádí ve **dvou** stranách pro **běžnou** buňku, což je znázorněno **modrými** šipkami. Ve **třech** stranách je to pro **rohové** buňky, což je znázorněno **červenými šipkami**. Mezi ně patří všechny ty, na kterých končí **černá čárkovaná šipka** pro danou stranu. Je to tedy bod, který je poslední v daném směru dané strany. Tři jsou z důvodu nalezení všech možných ploch. Všechna vyhledávání jsou závislá na směru průchodu spirálou. Na obrázcích 23 a 24, jsou zobrazeny situace, kdy v závislosti na zvětšení ***master*** modulu, je ovlivněn modul a ten se v původní velikosti do mřížky nevejde. Je tedy nutné změnit mu velikost v závislosti na volných plochách, které byly tímto algoritmem nalezeny.



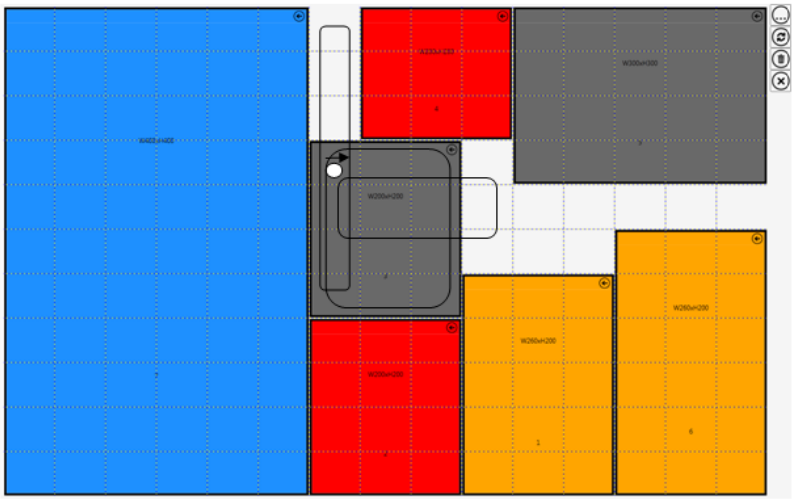
Obrázek 23: Příklad nalezení nového umístění ***slave*** modulu před změnou velikosti ***master*** modulu

Obrázek 23 zobrazuje původní konfiguraci. Nyní je zvětšena výška **červeného** modulu s číslem **dva** o jeden řádek nahoru. To má za následek, že **šedý** modul s číslem **tři** nelze v jeho původní velikosti nikam přesunout. Je tedy nutné pro tento modul nalézt nové místo co nejblíže jeho původnímu. Je tedy určena **startovací pozice**, od které se začne **spirálově** hledat. Její nalezení se věnuje kapitola 4.4.4. Na obrázku 24 jsou zobrazeny všechny nalezené pozice a použití té nejvhodnější.

Jelikož nalezená startovací pozice označená **bílým kolečkem** je první buňkou **spirálového vyhledávání**, hledá se ve směru **vpravo**. Ten je zobrazen **černou šipkou**. Toto tedy na obrázku 24 znamená, že se budou prohledávat plochy ve stranách **nahoru** a **doprava**. Navíc se jedná o **rohovou** buňku, takže přibude směr i **dolů**. Díky tomu algoritmus nalezl tři možné plochy. První má pozici <**0,6**> a velikost <**7,1**>, kde se od **startovací pozice** hledá **nahoru** a **dolů**. Zde bylo omezeno hledání **červenými moduly**. Druhá plocha má pozici <**3,6**> a velikost <**3,4**> a je omezena **šedým**, **červeným** s číslem **čtyři** a **oranžovým** s číslem **jedna**. Poslední s pozicí <**3,6**> a velikostí <**4,3**> je omezen oběma **červenými** moduly a **oranžovým** s číslem **jedna**. Pro tyto dvě poslední plochy se hledalo **doprava** a **dolů**. Všechny tyto tři nalezené plochy jsou vstupem pro **optimalizační** algoritmus z kapitoly 4.4.5, který z nich vybere tu nejvhodnější. V tomto případě byla určena jako nejlepší s pozicí <**3,6**> a velikostí <**4,3**>.

4.4.4 Nalezení startovací pozice

Jedním z posledních důležitých algoritmů je nalezení **startovací pozice**. Ta se využívá především při vkládání nového modulu do systému. Tato pozice se odvíjí od parametrů, kterými jsou počet sloupců a řádků tvořící mřížku. Vždy je snaha o to, aby v tomto případě byla **startovací pozice** vždy co nejblíže středu mřížky. Výpočet této pozice je velmi jednoduchý. Jednotlivá čísla se vydělí dvěma. Výsledek se zaokrouhlí nahoru a



Obrázek 24: Příklad nalezení nového umístění ***slave*** modulu po změně velikosti ***master*** modulu

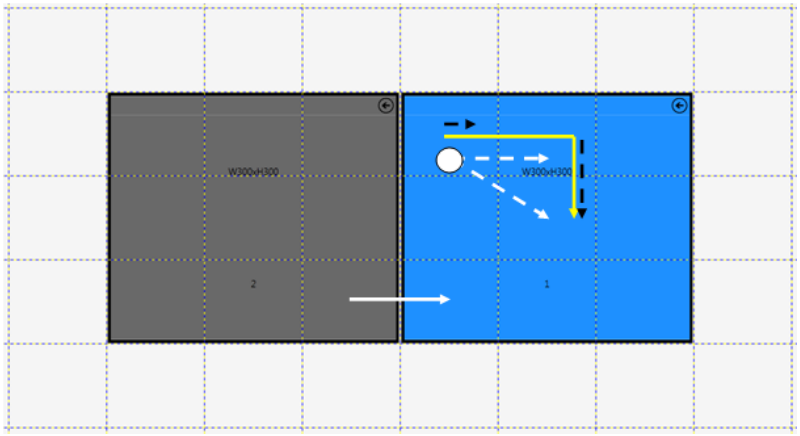
odečte se jedna. Ta se odebírá z důvodu, že pozice v mřížce se indexují od nuly. Pokud je vstupní parametr lichý, výsledná pozice je vždy na středu. Pokud je sudý, nelze se do středu přímo trefit a tato pozice je potom vždy blíže horní nebo levé hraně mřížky.

Druhého algoritmu pro nalezení této pozice se používá opět při hledání nového umístění či velikosti ***slave*** modulu, při změně pozice nebo velikosti ***master*** modulu. Opět se zde využívá principu **spirálového** vyhledávání. Je tu snaha najít nejbližší volnou buňku, od umístění té původní.

Princip spočívá v postupném procházení jednotlivých buněk daného směru od původní pozice modulu. Pro každý směr se jednotlivé volné buňky ukládají. Když se algoritmus dostane na **rohový** bod, zkontroluje, zda nějaké takové pozice našel. Pokud nenašel nebo **startovací** bod je **rohový**, tak pokračuje hledáním v dalším směru. Pokud však ano, tak se tyto položky vyhodnotí. Nalezne takovou buňku, jejíž vzdálenost mezi ní a původním umístěním je **nejkratší**. To samé se provádí, pokud se dostane do pozice, která je mimo grid. Výpočet vzdálenosti se provádí pomocí jednoduchého vzorce, pro výpočet vzdálenosti mezi dvěma body.

Na obrázku 25 je zobrazena situace před hledáním nového **startovacího** bodu. Jedná se o posunutí **šedého** modulu s číslem **dva doprava** o jednu pozici, to znázorňuje **bílá šipka**. To má za následek, že je nalezen jeden ***slave*** modul. Tedy **modrý** s pořadovým číslem **jedna**. Aby bylo možno tento posun realizovat, je nutné nejprve ověřit, zda je možno tento ***slave*** modul někam přesunou či zmenšit. Jelikož jde o posun **šedého** modulu doprava, překryje se tím původní **kotvící** pozice **modrého** modulu. Musí se najít nová **startovací** buňka, od které se začne **spirálové** hledat i nová pozice tohoto modulu.

Tyto pozice jsou znázorněny **bílými čárkovanými šipkami**. Byly tedy nalezeny dva možné **startovací body** pro **spirálový** průchod po **pravé** straně. Směry jsou vyznačeny **černými čárkovanými šipkami** a **spirála** pak **žlutou**.



Obrázek 25: Příklad nalezení startovací pozice

Posledním krokem je nalezení **nejkratší vzdálenosti** mezi původním kotvícím bodem a jedním z nalezených. To se realizuje pomocí následujícího vzorce.

$$\sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$$

Kde proměnné <a_1, a_2> znázorňují **původní** pozici a <b_1, b_2> pak ty **alternativní**. Původní **kotvící** pozice měla souřadnice <1,4> a **alternativní** body mají pozice <1,5> a <2,5>. Pokud je tedy do tohoto vzorce dosazeno, tak pro buňku <1,5> je vzdálenost jedna a pro <2,5> je vzdálenost odmocnina ze dvou, což je rozhodně větší než jedna. Je tedy jasné, že novým startovacím bodem pro další hledání je bod <1,5>.

4.4.5 Volba nejvhodnější plochy a optimalizace její velikosti

Poslední důležitou funkcí je volba **nejvhodnější plochy** a její **optimalizace**. Nalezení těchto ploch bylo vysvětleno v kapitole 4.4.3. Tento algoritmus postupně prochází jednotlivé plochy a nastavuje příslušnému modulu v závislosti na **šířce** a **výšce nejvhodnější stav**. Poté se zjišťuje, zdali je tato plocha nalezené položky menší než **minimální** velikost příslušného modulu.

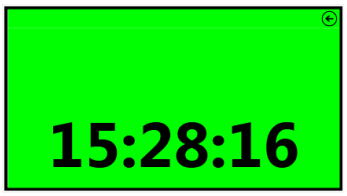
Dále se pak z jednotlivých konfigurací tohoto modulu nalezne ta s **nejvyšší** váhou. Pokud nastane situace, že se váhy shodují, bere se ta největší plochu.

Pokud se nějaká taková najde, algoritmus ji bude zpracovávat dalším krokem. Tím je změna **šířky** nebo **výšky** nalezené plochy tak, aby **poměrově** odpovídaly **minimální** velikosti aktuálně nalezeného **stavu**. Tedy pokud má aktuální stav minimální velikost 100x100 a nalezená plocha velikosti 130x150, tak se šířka změní z hodnoty 150 na 130.

4.5 Funkcionalita a stavy konkrétních modulů

Tato část popisuje stručně funkce jednotlivých implementovaných **funkčních** modulů. Dále jsou zde zobrazeny jejich jednotlivé **stavy**. Každý modul je tvořen vždy třemi stavy,

Obrázek 26: Modul *Time* v minimálním stavu a vahou 1



Obrázek 27: Modul *Time* ve stavu a vahou 2

jejichž velikost a funkcionalita se odvíjí od dat, která chtějí prezentovat. Další jejich společnou vlastností je, že jejich **minimální stav**, tedy stav s nejnižší vahou neprezentuje žádné informace, ale zobrazuje pouze ikonu, jejíž vzhled odpovídá dané funkcionalitě celého modulu. Architektura jednotlivých funkčních modulů je navržena s využitím návrhového vzoru **Model View ViewModel**, jehož širší kontext je v příloze B.3.[7]

4.5.1 Modul Time

Tento modul se věnuje prezentaci informací týkajících se času. Jeho funkce je tedy velmi jednoduchá. Na obrázku 26 je možno vidět modul v minimálním stavu.

Tento stav neprezentuje žádné informace. Zobrazuje pouze ikonu času. Jeho váha má hodnotu jedna a minimální velikost je (190, 190). Další dva stavy se od sebe liší pouze v tom, že ten s vahou tři na obrázku 28 zobrazuje i **datum**. Stav s vahou dva je na obrázku 27.

Obrázek 28: Modul *Time* ve stavu s vahou 3

Obrázek 29: Modul *PhotoDirectory* v minimálním stavu a vahou 1

Obrázek 30: Modul *PhotoDirectory* ve stavu a vahou 2

4.5.2 Modul PhotoDirectory

Druhým **funkčním** modulem je **PhotoDirectory**. Jeho funkcí je prezentovat uživateli obrázky v různých formátech. Úložiště těchto snímků je konfigurovatelné. Na obrázku 29 je možno vidět jeho minimální stav.

Nastavení cesty k adresáři s obrázky se provádí pomocí atributu *Path* v pomocném souboru toho modulu. Ten je uložen v adresáři *PhotoDirectory*. Název souboru je *temp.xml*. Tento adresář je uložen ve složce, ze které je celá aplikace spouštěna. Pokud tento adresář nebo příslušný soubor neexistuje, systém si jej sám automaticky vytvoří. Struktura souboru je velmi jednoduchá, obsahuje totiž pouze kořenový element *PhotoDirectory* a v něm jsou příslušné atributy. Stav s vahou dva v závislosti na zvoleném adresáři nalezne všechny obrázky s příslušnou příponou a náhodně je pak v jistém intervalu zobrazuje uživateli. Díky tomu vznikl druhý atribut. Ten se jmenuje *RandomPhotoInterval*. Na základě čísla do něj vloženého určuje periodu, jak často se bude obrázek měnit. Jeho hodnota je v sekundách. Pokud tento interval není zvolen, je načtena defaultní hodnota, které činí deset sekund. Na obrázku 30 je možno tento stav vidět.

Poslední stav tohoto **funkčního** modulu využívá pouze konfigurační informace o cestě k adresáři. Slouží uživateli k postupnému procházení jednotlivých obrázku z adresáře. Ten se může v této složce postupně posouvat nezávisle na čase. Jednotlivé posuny je

Obrázek 31: Modul *PhotoDirectory* ve stavu s vahou 3

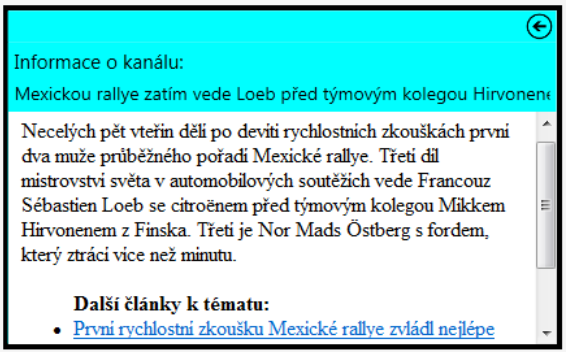
možno ovládat pomocí dvou tlačítek **vpřed** či **vzad**. Dále uživateli zobrazuje název aktuálně zobrazovaného a jeho pořadí. Taktéž i celkový počet nalezených obrázků. Pokud se uživatel dostane na konec či začátek, je příslušné tlačítko deaktivováno a nelze je použít. Jeho váha je tři a je ho možno vidět na obrázku 31.

4.5.3 Modul RssReader

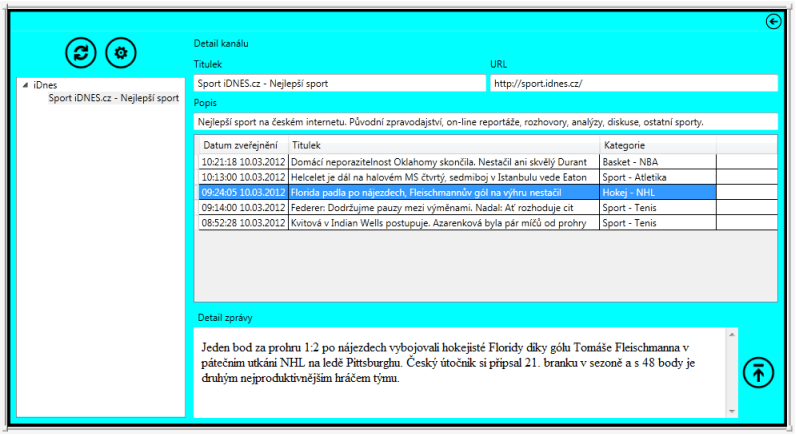
Jde tu o poslední z těchto **funkčních** modulů. Slouží pro prezentaci informací z *RSS* kanálů. Tyto si uživatel může sám přidávat či odebírat. Jedná se o nejkomplikovanější modul v systému. Je opět uživatelsky konfigurovatelný prostřednictvím příslušného souboru, který je uložen ve složce *RssReader*. Na obrázku 32 je zobrazen minimální stav s váhou jedna.

Další stav má váhu dva. Opět prezentuje stručné informace. V pomocném souboru jsou uloženy uživatelem definované skupiny a v nich jednotlivé *RSS* kanály. Tento soubor je v tomto stavu hojně využíván a má poněkud komplikovanější strukturu. Tvoří ji kořenový element s názvem *Root*. Ten může obsahovat atributy *AutoRefresh* a *TopMessages*. Dále pak může obsahovat několik vnořených elementů s názvem *Group*. Ty prezentují jednotlivé skupiny kanálů a jsou tvořeny atributy pro jejich názvy a jednoznačnou iden-

Obrázek 32: Modul *RssReader* v minimálním stavu a vahou 1



Obrázek 33: Modul *RssReader* ve stavu a vahou 2



Obrázek 34: Modul *RssReader* ve stavu s vahou 3

tifikaci. Každá taková skupina může pak obsahovat dále několik elementů *Channel*, které prezentují jednotlivé *RSS* kanály. Ty opět obsahují název, identifikaci a navíc adresu kanálu. Tento stav tedy dle hodnoty atributu *AutoRefresh* náhodně načítá jednotlivé kanály, které jsou v tomto souboru definovány. Z nich pak dále náhodně vybírá jednotlivé zprávy. Každá je pak uživateli viditelná po dobu, která je definována v tomto atributu. Pokud není doba nastavena, bere se automaticky hodnota třicet sekund. Hodnota atributu je opět v sekundách. Pokud nebyl nalezen žádný kanál, zobrazuje se uživateli pouze informace o tom, že nebyl nalezen žádný kanál. Tento stav je na obrázku 33.

Poslední stav má váhu tři. Slouží k lepšímu uživatelskému přístupu k jednotlivým kanálům. Uživatel zde může definovat jednotlivé skupiny a do nich přidávat konkrétní *RSS* kanály. Dále si zde může nezávisle prohlížet konkrétní zprávy z jednotlivých kanálů. Definuje také atributy pro interval načtení další zprávy předchozího stavu. Tento stav je zobrazen na obrázku 34.

Zde je možno stručně vysvětlit jednotlivé části. Vlevo je vidět takzvaný **navigátor**. V něm jsou do stromu načteny jednotlivé skupiny a pod nimi příslušné kanály. Ty je možno přidávat, upravovat nebo odebírat. Stejně jako skupiny a to pomocí pravého tlačítka nad

Obrázek 35: Struktura aplikace *CleverGridApplication*

konkrétní skupinou či kanálem. Nahoře jsou dvě tlačítka jedno pro refresh navigátoru, druhé pro nastavení intervalu automatického refreshe vybraného kanálu a další pro zobrazení počtu nejnovějších zpráv. Vpravo jsou zobrazeny konkrétní informace o vybraném kanálu. V hlavičce jsou základní informace a v tabulce pod ní jsou jednotlivé zprávy. Tato tabulka je ovlivněna atributy, které se nastavovaly v navigátoru. Tedy aktuálně je nastaveno zobrazování pěti nejnovějších zpráv a tato tabulka se bude aktualizovat každých čtyřicet vteřin. Dole je detail konkrétní zprávy a pomoci tlačítka vpravo je možno si tuto zprávu zobrazit úplně.

4.6 Logické rozložení projektu

Projekt je velmi rozsáhlý a bylo nutné jej nějakým rozumným způsobem logicky rozdělit. Nejvhodnějším způsobem bylo rozdělení podle určitých společných vlastností do binárních knihoven. Dalším důvodem rozdělení bylo i to, že některé funkční vlastnosti se neustále opakovaly a bylo tedy zbytečné neustále stejné zdrojové kódy v aplikaci kopírovat. Celý projekt je tedy rozdělen na **dvě základní části**. První nese název *CleverGridApplication*. Tvoří **jádro** celé práce a týká se především **inteligentní mřížky** a problémů s ní spojených. Je to hlavní a spouštěcí část celého projektu. Toto rozložení je možno vidět na obrázku 35. Druhá část nese název *SocialService* a je popsaná v kapitole 5.1.3, která se celá zabývá sociálním chováním aplikace.

Hlavní část je tvořena čtyřmi základními celky. Každý z těchto celků řeší vlastní problematiku a má svůj důvod, proč je oddělen.

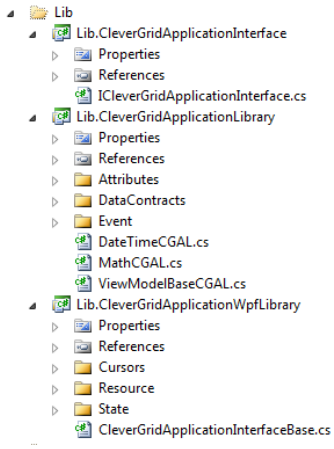
- **Lib**

  Tento celek je tvořen třemi projekty. Jedná se o jakési pomocné projekty, které jsou velmi často používány na různých místech v aplikaci. Část *Lib* je vidět na obrázku 36.

  - ***Lib.CleverGridApplicationInterface***

    Poskytuje **rozhraní**, které musí implementovat každý modul použitý v aplikaci. Rozhraní reprezentuje tedy pravidla, na základě kterých je aplikace schopna rozpoznat, zda jde o modul, s kterým je systém schopen korektně komunikovat a předávat si informace.
  - ***Lib.CleverGridApplicationLibrary***

    Obsahuje především třídy, se kterými systém často spolupracuje a využívá opakovaně jejich metody. Patří sem například funkce pro zaokrouhlování čísel



Obrázek 36: Struktura projektu *Lib*

nahoru či dolů. Třída *ViewModelBaseCGAL* tvoří základ při tvorbě většiny formulářů, které byly navrženy pomocí modelu *MVVM*, jehož detailní popis je v příloze B.3. Dále obsahuje **atribut**, na základě kterého aplikace rozpoznává, zda je modul určen pro tuto aplikaci. Předposledním blokem této knihovny jsou takzvané "Datakontrakty". Ty slouží pro přenos informací o pozicích a velikostech jednotlivých modulů prostřednictvím *WCF služby*. Tato služba tyto informace zpracovává pro další použití týkající se sociálního chování modulů. Poslední složkou je *Event*, která slouží pouze pro zaobalení potřebných informací přenášených prostřednictvím speciálních události vyvolávaných v systému.

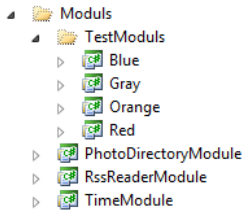
– ***Lib.CleverGridApplicationWpfLibrary***
  Jedná se o poslední pomocnou knihovnu. V ní je především uloženo rozhraní, které umožňuje jednotlivým modulům přepínat se mezi svými implementovanými stavy. Dále pak *WaitCursor*, který uživatele informuje, že systém zrovna pracuje. Poslední velmi důležitou třídou je *CleverGridApplicationInterfaceBase*. Tato třída tvoří základ všem modulům, které byly implementovány. Poskytuje jim všechny společné a pomocné vlastnosti a funkce, které dané moduly nutně využívají.

- **Moduls**

  Blok se týká samostatných modulů a struktura je zobrazena na obrázku 37. Obsahuje složku *TestModuls*,ve které jsou obsaženy všechny čtyři implementace **testovacích** modulů, které byly použity v systému. V kořeni hlavní složky jsou obsaženy tři projekty, jež reprezentují jednotlivé **funkční** moduly.

- **Proxy**

  Jelikož systém v současném stavu potřebuje komunikovat pouze s jedinou službou, je blok *Proxy* na obrázku 38 tvořen pouze jedním projektem. Tato knihovna má pouze



Obrázek 37: Struktura složky *Moduls*

Obrázek 38: Struktura složky *Proxy*

jednoduchou funkci. Tvoří jakéhosi prostředníka mezi klientem, tedy systémem a službou, kterou představuje server. Jejím úkolem je tedy zprostředkovávat spojení mezi těmito dvěma body.

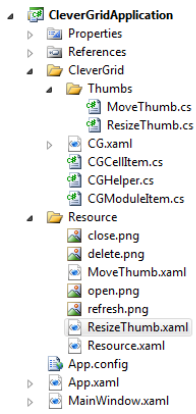
- **CleverGridApplication**

  Jde tu o poslední projekt hlavní části aplikace, který je vůbec nejdůležitějším. Tvoří bránu a poskytuje přístup ke všem funkcím systému. Obsahuje jak vlastní grafickou prezentaci **inteligentní mřížky**, tak i úplnou implementaci všech funkcí a algoritmů, které jsou nutné k její realizaci. Využívá většiny předešlých knihoven. Tento projekt je také **hlavním spouštěcím blokem** celé aplikace.

  Obsahuje ale i další velmi důležité prvky. V adresáři *CleverGrid* jsou všechny základní potřebné objekty. Jsou zde obsaženy třídy, které realizují komunikaci mezi uživatelem a **inteligentní mřížkou**. Jsou v nich uloženy také informace o volných či obsazených buňkách **inteligentní mřížky**. Dále pak nesou informace o použitých modulech.

  Další důležitou složku tvoří *Thumbs*. Zde jsou uloženy prvky, které odchytávají potřebné události. Na základě těchto prvků je systém schopen odlišovat chování modulů a zjistit, zda se jedná o přesun nebo změnu velikosti aktuálního modulu.

  V poslední složce *Resource*, která je uložena v kořeni tohoto projektu, jsou obsaženy informace o jednotných stylech použitých v aplikaci. Jsou zde i styly pro ovládací prvky každého modulu. Složka s tímto názvem je obsažena u většiny projektů, které poskytují grafickou prezentaci, tedy projekty které obsahují nějaké uživatelské rozhraní. Typickým příkladem jsou projekty, ve kterých se realizují implementace modulů použitých v systému. Na obrázku 39 je možno vidět strukturu tohoto projektu.



Obrázek 39: Struktura projektu *CleverGridApplication*

5 Pokročilé možnosti

Kromě základních funkcí jako je práce s moduly, ovládání jejich chování a dalších, aplikace poskytuje i některé pokročilé možnosti. Mezi ně zajisté patří sbírání, zpracovávání a vyhodnocování **sociálních aspektů**. V systému jsou některé základní funkce pro práci s těmito informace implementovány. Mezi další patří například oblast, která se týká problematiky **multidoteků**.

5.1 Sociální prvky

Mezi tyto prvky patří různé druhy informací. Aplikace umí některé z nich hlídat, uchovávat je a zpracovávat. Mezi takové informace patří například seznam všech použitých modulů, které byly do aplikace vloženy. Každá instance této aplikace tedy může používat různé moduly. Existuje tedy jedno místo, se kterým všechny tyto aplikace komunikují a tyto informace mu předávají k dalšímu zpracování. Mezi další takové možné informace patří umístění a velikosti jednotlivých modulů. I tato data je systém schopen hlídat a ukládat. Tato aplikace tedy uchovává vybrané informace neboli **sociální prvky** v jednotném úložišti. Tato data je pak možné filtrovat a dále analyzovat. Aplikace tuto funkci podporuje a je schopna v těchto datech provést jisté základní výběry. Ty pak nabízí uživateli. Mezi tyto informace patří například nejpoužívanější velikost a umístění konkrétního modulu.

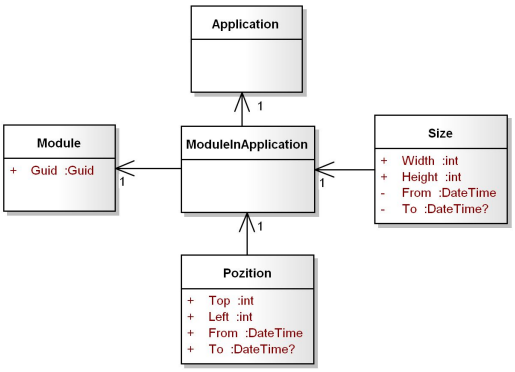
5.1.1 Způsob komunikace

Pro řešení komunikace bylo nejvhodnější použít architekturu *klient-server*. Zároveň bylo nutné od sebe oddělit jednotlivé části aplikace. Systém, který zpracovává a řídí "práci" modulů, tedy *CleverGridApplication*, tvoří jednu část a bude **klientem**. Dále se bude používat název **klientská aplikace**. Podrobné informace o této architektuře jsou popsány v kapitole 4.6. Druhá část, bude jednotlivé informace zpracovávat, uchovávat a poskytovat uživateli. Jedná se tedy o **server**. Pro vlastní komunikaci a výměnu dat, mezi těmito dvěma prvky, bylo nejvhodnější použít technologie **webových služeb** a to konkrétně *WCF*. Bylo tedy potřeba vytvořit druhý projekt, který tyto služby bude poskytovat. Ten se jmenuje *SocialService*. Ten pak spolupracuje s databází s názvem *Social_behavior*.

5.1.2 Práce se sociálními aspekty v aplikaci

Systém umožňuje dvě základní funkce pro práci se **sociálními prvky**. První z nich je ukládání informací o aplikaci, modulu, jeho pozici a velikosti. Druhou je základní vyhodnocení těchto dat a zaslání příslušné odpovědi klientovi. Pro co nejjednodušší práci je navržena struktura, která je na obrázku 40.

Jedná se o obraz databázových tabulek. Postupně bude nyní vysvětleno, proč a k čemu jsou jednotlivé prvky vhodné. Základním článkem tohoto diagramu je objekt *Application*, který velmi jednoduchým způsobem identifikuje konkrétní instanci aplikace. Každá z nich je označena číslem. Její registrace se provádí při prvním startu



Obrázek 40: Struktura pro uchovávání sociálních prvků

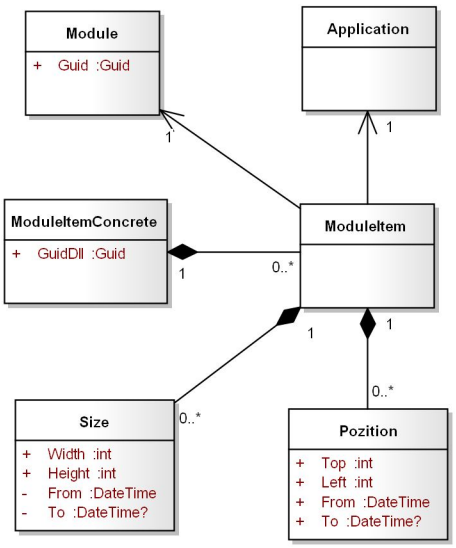
klienta, kdy se dívá do registru, jestli tam má uloženou identifikaci aplikace. Je tedy zkontrolováno v kořeni registrů pro aktuální uživatele, zdali existuje hodnota pro klíč ***CleverGridApplication\\CGApplicationId***. Pokud tato hodnota neexistuje, je zaslán požadavek na server o registraci aplikace. Ten vytvoří nový záznam v *Application* a vrátí jej klientovi. Ten si tuto hodnotu uloží do registru a může s ní pak pracovat. Pokud nějaká hodnota existuje, provede se na serveru ověření, zda je uložena i v databázi. Pokud ano, je navrácena. Pokud neexistuje, tak ji server uloží. Takto se tedy registruje každý klient po startu.

Dalším důležitým objektem je *Module*. Ten zastupuje jednotlivé moduly. Má jedinou položku, která jednoznačně identifikuje typ modulu a odpovídá položce *ModuleGuidDll* v rozhraní na výpisu 9 v kapitole 4.2.1.2.

Dalším objektem je *Size*, který zastupuje velikost modulu. Má položky *Width* a *Height*, které odpovídají šířce a výšce modulu. Dále pak položky *From* a *To*. Ty znamenají, jak dlouho měl daný modul danou velikost. Obdobně je tomu i u objektu *Pozition*, kdy tyto položky mají stejný význam pro pozici. *Top* a *Left* slouží pro identifikaci umístění daného modulu na *Canvasu*. Tyto dva objekty se vztahují vždy ke konkrétnímu modulu v konkrétní instanci aplikace. To je uchováváno v objektu *ModuleInApplication*.

Aplikace poskytuje dvě základní funkce. První z nich je získávání informací o pozicích a velikostech jednotlivých modulů. Tato data se zapisují do speciální pomocné struktury vždy, pokud dojde k některé z těchto událostí. Tato struktura je na obrázku 41. Druhou funkcí je opačný směr. Tedy zpracování těchto informací a předání klientovi. Toho se využívá při načítání nového modulu, kdy aplikace pošle serveru identifikaci modulu, který chce vložit do mřížky. Server díky tomu vyhledá příslušný modul a vrátí jeho nejpoužívanější pozici a velikost. Na základě těchto dat klient umístí tento modul co nejblíže vyhledané pozici a s danou nebližší možnou velikostí. Tuto funkci je opět možno vypnout pomocí klíče *LoadInfoFromService*. Celkovou komunikaci se službou je možno vypnout či zapnou pomocí klíče *SocialServiceEnabled*.

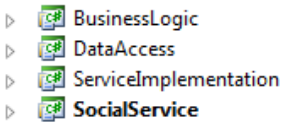
Na základě uložených dat je možno provádět velké množství kombinací při vyhledávání mezi nimi. K nim patří například:



Obrázek 41: Paměťová struktura pro dočasné uchovávání a přenos sociálních prvků

- Vyhledání nejpoužívanějšího nebo nejméně používaného modulu v závislostech na
  - Čase
  - Aplikaci
  - Pozici
  - Velikosti
- Vyhledání optimální velikosti či pozice modulu
- Vyhledání, kolik aplikací daný modul používá nebo nepoužívá
- Jestli je daný modul v aplikaci použit vícekrát než jednou
- Jak často aplikace bývá spuštěna
- V jakém časovém horizontu aplikace bývá nejčastěji používána
- . . .

Tedy například pokud se přidá nový modul, okamžitě se zapíše informace o jeho aktuální pozici a velikosti pro čas, kdy byl do aplikace vložen. Pokud je pak tento modul někam přemístěn, do struktury je zapsána informace, že modul na této pozici byl do jistého času a vytvoří se nový záznam s tímto časem a novou pozicí. Ukládání těchto informací se provádí vždy při přidání nového nebo náhodného načtení modulů, odebrání konkrétního nebo všech modulů, posunu nebo změny pozice ***master*** modulů a následné změny ***slave*** modulů. Pomocná struktura je pak v jistém časovém intervalu zasílána na server, kde jsou jednotlivé informace ukládány. Tento interval je možno nastavit v konfiguračním souboru



Obrázek 42: Struktura serveru *SocialService*

pod klíčem *SaveInfoToServiceInterval*. Dále je pak možné toto ukládání vypnout pomocí klíče *SaveInfoToService*. Velikost a pozice se vždy zaokrouhlí dle klíče *NearestNumber*. Tedy pokud bude jeho hodnota 50 a výška modulu bude 130, tak se nastaví na 150. Je z toho důvodu, že každá aplikace může mít různou konfiguraci sloupců a řádků. Díky tomu i různé šířky a výšky buněk. Systém se pomocí tohoto zaokrouhlovaní snaží data co nejvíce sjednocovat a ulehčí si tak práci při jejich zpracovávání.

5.1.3 Návrh architektury

Jelikož se jedná o návrh struktury pro server, byla použita takzvaná **třívrstvá architektura**. Jde tu o rozdělení projektu do tří základních částí, z nichž každá samostatně plní určitou důležitou funkci. Rozložení projektu je možno vidět na obrázku 42.

SocialService Je hlavním spouštěcím projektem nejenom vlastního serveru, ale také všech služeb, které tato aplikace poskytuje. Tento projekt komunikuje s nejvyšší vrstvou. Tou je *ServiceImplementation*.

ServiceImplementation Implementuje jednotlivá rozhraní, která daná služba poskytuje. Další důležitou funkcí je, že otevírá příslušné spojení s databází. Komunikuje s další vrstvou, která nese název *BusinessLogic*.

BusinessLogic Stará se o veškerou logiku. Tedy o různé transformace dat a jejich další zpracovávání. Dále pak připravuje tato data pro uložení do databáze.

DataAccess Je to poslední a nejnižší vrstva architektury. Přímo komunikuje s databází a jsou v ní uloženy všechny dotazy na ni. Její výhodou je, že pokud se změní databáze, vymění se pouze tato vrstva a všechny nadřazené funkce fungují beze změny.

5.2 Multidoteky

Aplikace byla realizována pomocí technologie *WPF*. Ta podporuje kromě standardního ovládání jednotlivých grafických prvků také možnost pomocí takzvaných **multidoteků** neboli **vícedotykové ovládání**. To je založeno na schopnosti snímacího zařízení vnímat více dotyků najednou. Například klasický *touchpad* u většiny notebooků je schopen vnímat pouze jeden položený prst, zatímco modernější zařízení, jako některé *tablety* či například *iPhone* nebo *iPad*, umí zpracovat více dotyků najednou.[8]

Windows 7 umožňují používat mutlidoteky ve třech režimech:

Obrázek 43: Ukázka multidoteků

"**Syrové dotyky**" Poskytuje přístup ke všem dotykovým zprávám. Je užitečný pro programy, které vyžadují přímý přístup ke všem primitivům a vlastní interpretaci a zpracování zpráv.

Gesta Abstrakce předchozího režimu. Interpretuje všechny nižší úrovně akce a převádí je do předem definovaného gesta. Mezi nejčastější gesta patří zoom a otáčení. Jedná se o snadný programovací model, který má však omezení. Tím je tendence řešit pouze jedno gesto v čase. Například otáčet nebo zoomovat.

Manipulace a setrvačnost Nadřazen nad gesta. Co jde udělat s gesty, jde v tomto režimu. Je to hlavně zisk větší manipulace a pružnosti.

WPF4 zahrnuje podporu pro dotyk a manipulaci. Tato podpora se vztahuje na celé platformě *UIElement*, *UIElement3D* a *ContentElement*. Všechny tyto elementy podporují události *TouchDown*, *TouchUp*, *TouchMove*, *TouchEnter* a *TouchLeave*. Všechny tyto události mají parametr *TouchEventArgs*, který poskytuje informace o *TouchDevice* a *TouchPoint*. Díky tomu je možno zjistit, jakým směrem byl pohyb, zda nahoru či dolů. Nebo zda šlo o přesun pomocí *TouchAction*. Vytvoření jednoduché aplikace, která podporuje mutlidoteky je realizováno třemi kroky.

- Nejprve je nutné na prvek, který má multidoteky podporovat nastavit hodnotu *IsManipulationEnabled* na *True*.
- V druhém kroku se nastaví po odchycení události *ManipulationStarting* mód, který je požadovaný. V mřížce by se jednalo o **zoom** a **přesun**.
- Posledním krokem je odchycení a zpracování události *ManipulationDelta*, díky které je možno například daný objekt **zvětšit**, **přesunout** či ho **otáčet**.

Na výpisu 12 je zobrazeno zpracování této události.

```
void image_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
  //získání zdroje
  var element = e.Source as FrameworkElement;
  if ( element != null )
```

```
    {
      //e.DeltaManipulation – došlo ke změně
      // Scale = zvětšení/zmenšení; 1.0 poslední(původní) velikost,
      // (1.1 == zvětšení 10%, 0.8 = zmenšení 20%)
      // Rotate = rotace
      // Translation = přesun
      var deltaManipulation = e.DeltaManipulation;
      var matrix  = ((MatrixTransform)element.RenderTransform).Matrix;

      // najdu původní(předchozí) střed
      Point center =  new Point ( element.ActualWidth/2, element.ActualHeight/2) ;
      center = matrix.Transform(center);

      //zoom
      matrix.ScaleAt(deltaManipulation.Scale.X, deltaManipulation.Scale.Y, center.X, center.Y);

      // rotace
      matrix.RotateAt(e.DeltaManipulation.Rotation, center.X, center.Y);

      //přesun
      matrix.Translate(e.DeltaManipulation.Translation.X, e.DeltaManipulation.Translation.Y);

      ((MatrixTransform)element.RenderTransform).Matrix = matrix;
      e.Handled = true;
    }
  }
```

Výpis 12: Příklad odchycení a zpracování události *ManipulationDelta* při práci s multidoteky

Při zpracovávání této události se pracuje s pozicemi v pixelech. Díky tomu by nebylo příliš obtížné tuto technologii zakomponovat do aplikace. Ta obsahuje algoritmy, které umí na základě těchto dat určit, jakým směrem bylo například přesouváno, dále pak zjistit, o kolik řádků či sloupců by se tato pozice v mřížce změnila. Obdobně jsou totiž řešeny funkce pro změnu pozice modulu v kapitole 4.4.1 nebo změnu velikosti z kapitoly 4.4.2.[9]

6 Závěr

Cílem této diplomové práce je návrh a realizace systému disponujícího jednoduchým uživatelským rozhraním, který je založen na modulární platformě. Uživatel může pracovat s konkrétním modulem a tím nezávisle ovlivňovat i jiné moduly. Ty se automaticky přemísťují na nejvhodnější pozice, díky algoritmům, které jsou založeny na technologii **spirálového vyhledávání**. Byly proto navrženy algoritmy, které se starají o jejich chování a jsou stěžejní pro tuto práci. Dále je poskytnuto rozhraní a návody, jak implementovat nové vlastní moduly, se kterými pak sytém umí jednoduše pracovat. Obsahuje i realizaci konkrétních modulů s množstvím funkcí pro prezentaci různých forem informací. Mezi důležitou funkci, kterou aplikace poskytuje je i návrh a realizace práce se **sociálními prvky**. Ty umí schraňovat a jednoduše vyhodnocovat. S tím souvisí chování jednotlivých modulů v systému. V neposlední řadě bylo použito aktuálně moderních technologií, které poskytují velké množství různých funkcí. Mezi nimi jsou různé možnosti ovládání uživatelského rozhraní. Aplikace je ovládána klasickým způsobem, ačkoliv je připravena na použití a jednoduchém zakomponování technologie **multidoteků**.

Aplikaci je samozřejmě možno dále rozšiřovat o další možné funkce. Mezi vybrané patří zajisté již poukázané zakomponování ovládání sytému pomocí technologie **multidoteků**. Jednou z další úprav je implementace návrhového vzoru *faktory*, pro využití různých typů algoritmů týkajících se vyhledávání volných pozic pro moduly. S tím souvisí rovněž **spirálové vyhledávání**, kde by bylo možno měnit jeho počáteční směr. Dalším možným rozšířením je úprava algoritmu pro optimalizaci stavů modulu a vyhledávání alternativních pozic ***slave modulů***.

Bc. Tomáš Cigánek

7 Reference

[1] Windows Developer Preview Windows 8 guide. *Microsoft Download Center* [online]. 2011, č. 1 [cit. 2012-02-19]. Dostupné z: http://download.microsoft.com/download/1/E/4/1E455D53-C382-4A39-BA73-55413F183333/Windows_Developer_Preview-Windows8_guide.pdf

[2] C# aplikace s podporou pluginů. *Programujte.com - web o programování, webdesignu, počítačové grafice, databázích, elektrotechnice a designu* [online]. 2006, č. 1 [cit. 2012-02-23]. Dostupné z: http://programujte.com/clanek/2006041802-c-aplikace-s-podporou-pluginu/

[3] MACDONALD, Matthew. , Jim a Ila NEUSTADT. *Pro WPF in C# 2010: Windows presentation foundation in .NET 4*. New York, N.Y.: Distributed to the book trade worldwide by Springer-Verlag, c2010, 1181 s. Expert's voice in .NET. ISBN 14-302-7205-8.

[4] *C#: programujeme profesionálně*. 1. vyd. Brno: Computer Press, 2003, 1130 s. ISBN 978-80-251-1503-9.

[5] PECINOVSKÝ, Rudolf. *Návrhové vzory*. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.

[6] ARLOW, Jim a Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. Vyd. 1. Překlad Bogdan Kiszka. Brno: Computer Press, 2007, 567 s. ISBN 978-80-251-1503-9.

[7] GAROFALO, Raffaele. *Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern*. Sebastopol, Calif: O'Reilly Media. ISBN 978-073-5650-923.

[8] Vícedotykové ovládání - Wikipedie. In: *Wikipedia* [online]. 2012-03-08 [cit. 2012-03-22]. Dostupné z: http://cs.wikipedia.org/wiki/Vícedotykové_ovládání

[9] Introduction to WPF 4 Multitouch. *MSDN Blogs* [online]. 2009-11-4, č. 1 [cit. 2012-03-22]. Dostupné z: http://blogs.msdn.com/b/jaimer/archive/2009/11/04/introduction-to-wpf-4-multitouch.aspx

[10] Objekty - Singleton Pattern. Objekty - *Objektová analýza, návrh a programování* [online]. 2005-06-16 [cit. 2012-03-03]. Dostupné z: http://objekty.vse.cz/Objekty/Vzory-Singleton

[11] Objekty - State Pattern. Objekty - *Objektová analýza, návrh a programování* [online]. 2005-06-16 [cit. 2012-03-04]. Dostupné z: http://objekty.vse.cz/Objekty/Vzory-State

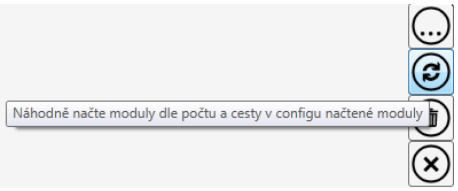
[12] MVVM: Model-View-ViewModel. *Programování* [online]. [cit. 2012-03-05]. Dostupné z: http://dajbych.net/model-view-viewmodel

[13] Poznáváme C# a Microsoft.NET 36. díl - úvod do reflexe. *O počítačích, IT a internetu - Živě.cz* [online]. 2005-08-12, č. 1 [cit. 2012-03-04]. Dostupné z: http://www.zive.cz/clanky/poznavame-c-a-microsoftnet-36-dil–uvod-do-reflexe/sc-3-a-126122/default.aspx

A Pomocné funkce

A.1 Funkce náhodného načtení modulů

Systém umožňuje funkci pro hromadné načtení určitého počtu modulů do systému. Tato funkce nemá žádný širší smysl. Je realizována spíše pro testovací účely. Je uživatelsky konfigurovatelná, takže uživatel může před spuštěním aplikace nastavit v konfiguračním souboru aplikace vlastní hodnoty potřebným parametrům a systém na to pak automaticky reaguje. Tyto parametry jsou celkem dva. První s klíčem *RandomLoadModulsPath* definuje cestu, kde budou jednotlivé knihovny představující moduly umístěny. Druhým je klíč *RandomLoadModulsCount*. Ten určuje kolik se má do aplikace modulů načíst. Realizace této funkce využívá algoritmu z kapitoly 4.2.1.1 pro načtení modulu. Tato funkce se volá v cyklu tolikrát, podle hodnoty nastavené v klíči *RandomLoadModulsCount*. Před spuštěním tohoto cyklu si aplikace načte všechny soubory z adresáře, jehož cesta je uložena v příslušném klíči. Poté v každém průchodu cyklu si načte jeden náhodný soubor z této složky. Pokud se jedná o modul, který lze do systému připojit, tak mu přidělí pořadí, ve kterém byl vložen. To pak inkrementuje. Pokud by se nejednalo o modul, tak v této iteraci zkouší připojit další náhodný soubor. To se opakuje, dokud jej nenajde. Na obrázku 44 je zobrazeno tlačítko z hlavního panelu aplikace, pomocí kterého je možno tuto funkci spustit.



Obrázek 44: Náhodné načtení modulů

B Návrhové vzory

B.1 Návrhový vzor *Singleton*

Tento návrhový vzor řeší v celku velmi jednoduchou úlohu. Poskytuje jednu jedinou instanci konkrétní třídy v rámci celého systému, které je pak přístupná všem objektům v tomto systému. Patří do skupiny pro tvorbu objektů. Zde patří například i vzor *Factory*. Obecné schéma vzoru je vidět na obrázku 45.[10]

B.2 Návrhový vzor *State*

Návrhový vzor řeší problém, jak změnit chování objektu, jestliže se změní jeho vnitřní stav. Po změně chování se objekt jeví jako instance jiné třídy. Na obrázku 46 je možno vidět obecné schéma tohoto vzoru.[11]

B.3 Návrhový vzor *Model View ViewModel*

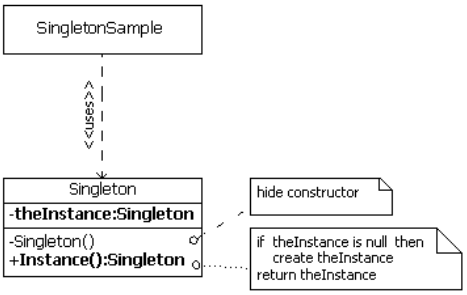
Jelikož implementace **funkčních** modulů byla výrazněji náročnější a jednotlivé moduly se skládají z více než jednoho formuláře, které mezi sebou komunikují, bylo nutné zde použít vhodnou architekturu. Tou je využití návrhového vzoru, který se nazývá *Model View ViewModel*, ve zkratce *MVVM*. Jedná se o vzor, který se velmi doporučuje využívat při tvorbě aplikací pomocí technologie *WPF*, *Silverlight* a dalších podobných.

Model-View-ViewModel je návrhový vzor pro *WPF* aplikace. Nabízí řešení, jak oddělit logiku aplikace od uživatelského rozhraní. Kódu je pak méně, vše je přehlednější a případné změny nejsou implementační noční můrou. *MVVM* odděluje data, stav aplikace a uživatelské rozhraní. Samotné *WPF* bylo vytvořeno tak, aby se v něm *MVVM* používal pohodlně. Proto vše pěkně doplňuje *binding* a *command*, tedy náhrada za uživatelské rozhraní řízené událostmi.

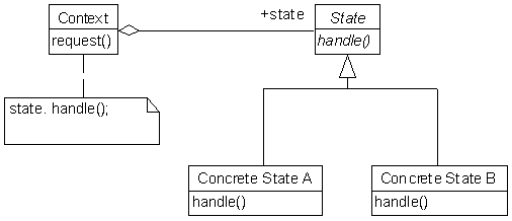
Složitost třídy roste s její chytrostí exponenciálně. Proto je výhodnější mít více hloupých tříd, než jednu chytrou. Pokud se přikloníme k dělení kódu do více tříd, nabízí se otázka, jak to provést správně. Jedním řešením je právě *Model-View-ViewModel*, který představuje vyzkoušené a ověřené řešení. Podle tohoto vzoru je naprogramován například *Expression Blend*.

- **Model**
  Obsahuje referenci na zdroj dat. Pokud obsahuje reference na více služeb z jednoho logického kontextu, nabízí celý kontext z více služeb v rámci jednoho celku, čili sebe. Naopak může být více modelů vedle sebe pro různé logické kontexty čerpající z jedné webové služby. Pokud se na straně klienta používá datová proxy, implementuje se do této vrstvy.

- **View**
  Zastupuje grafické rozhraní v jazyce *XAML* s troškou nezbytného kódu *C#* na pozadí, který především provádí vytvoření nové *ViewModel* třídy či tříd, za kterých jednotlivé formulářové prvky čerpají svůj obsah. *Binding* skrz *ObservableCollection*



Obrázek 45: Návrhový vzor *Singleton*



Obrázek 46: Návrhový vzor *State*

umožní automatickou změnu obsahu ovládacího prvku při změně obsahu této datové struktury. A naopak, změní-li uživatel obsah ovládacího prvku, projeví se to i v datové struktuře. Nově přidané či odebrané prvky jsou rychle k nalezení v obsluze události, kterou tato změna vyvolá. Je samozřejmě možné využívat i jiné datové struktury s vlastními ovládacími prvky. Při používání tříd pro *binding* se v *XAML* kódu jen jednoduše deklaruje, která vlastnost třídy se má do obsahu ovládacího prvku vkládat. Je tedy možné velice pohodlně čerpat z více ovládacích prvků různé vlastnosti jedné třídy. Pokud třída implementuje rozhraní *INotifyCollectionChanged* a *INotifyPropertyChanged*, projeví se změny na obou stranách (uživatelského rozhraní a datové struktury) automaticky.

- **ViewModel**
  Spojuje *Model* a *View*. K *Modelu* se neváže přímo, ale přes rozhraní, které Model implementuje. Konkrétní *Model* se předá v konstruktoru. Je dovoleno mít i bezparametrický konstruktor, který vytvoří výchozí instanci *Modelu*. Ovládací prvky provádějí *binding* z této třídy. V této třídě se provádí filtrování dat v závislosti na stavu ovládacích prvků. Model nesmí o stavu ovládacích prvků nic vědět.[12]

C Implementace vybraných algoritmů

C.1 Algoritmus implementace návrhového vzoru *Singleton*

```
private CGHelper(Grid p_Grid)
{
  _Grid = p_Grid;
  Init();
}

private static CGHelper _CGHelper;
public static CGHelper GetInstance()
{
  if (_CGHelper == null)
  {
    throw new Exception(@"Nebyla inicializována proměnná '_CGHelper' pomocí metody '
        GetInstance(Grid p_Grid)'");
  }
  return _CGHelper;
}

public static CGHelper GetInstance(Grid p_Grid)
{
  if (_CGHelper == null)
  {
    _CGHelper = new CGHelper(p_Grid);
  }
  return _CGHelper;
}
```

Výpis 13: Algoritmus implementace návrhového vzoru Singleton

V algoritumu 13 existuje privátní statická proměnná s názvem *_CGHelper*, dále pak privátní kontraktor s parametrem, který je tytu *Grid*. To proto, že je potřeba získat instanci této komponenty a díky tomu s ní možno dále pracovat. Poslední podmínkou byla statická metoda, která vrací jedinou instanci. Zde je vidět, že tyto metody jsou dvě. To je z jednoduchého důvodu. Jelikož je potřeba dále pracovat s instancí komponenty *Grid*, je nutné si ji předat. To se děje při spuštění aplikace, kdy třída *CG*, která obsahuje přímo konkrétní instanci komponenty *Grid*, zavolá metodu *GetInstance* s parametrem, kde předá tento *Grid*. Tato metoda pak jenom zjistí, jestli neexistuje již instance třídy *CGHelper*, pokud ne, tak ji vytvoří a následně vrací tuto instanci. Metoda *GetInstance* bez parametru pak již předpokládá, že při spuštění aplikace se tato instance vytvořila a pouze ji vrací.

C.2 Algoritmus odebrání modulu

```
public void Module_RemoveModuleClick(object sender, EventArgs e)
{
  if (MessageBox.Show("Opravdu chcete vybraný modul odebrat?",
                      "Odebrání modulu", MessageBoxButton.YesNo) == MessageBoxResult.Yes)
  {
```

```
    ContentControl cc = ((sender as UserControl).Parent as ContentControl);

    int row = GetRow(cc);
    int column = GetColumn(cc);
    int rowSpan = GetRowSpan(cc);
    int columnSpan = GetColumnSpan(cc);

    _Grid.Children.Remove(cc);

    CGCellItem item = GetItemByPozition(row, column);

    Moduls.Remove(item.CurrentItem);

    SetItemToMatrix(_Matrix, row, column, rowSpan, columnSpan, null);
  }
}
```

Výpis 14: Algoritmus odebrání modulu

C.3 Algoritmus nastavení stylu pro příslušný mód

```
public void Module_ResizeClick(object sender, EventArgs e)
{
  ContentControl cc = ((sender as UserControl).Parent as ContentControl);

  if (!( e as EventArgsBase).IsInResizeMode)
  {
    SetNormalMode(cc);
  }
  else
  {
    SetCurrentMode(cc, "PresenterItemTemplateResize");
  }
}
```

Výpis 15: Algoritmus nastavení stylu pro příslušný mód

C.4 Algoritmus odchycení a zpracování události *DragDelta* pro změnu pozice *master* modulu

```
private void MoveThumb_DragDelta(object sender, DragDeltaEventArgs e)
{
  _CurrentItem = this.DataContext as Control;
  if (_CurrentItem != null)
  {
    _Changed = false;
    _Left = Canvas.GetLeft(_CurrentItem);
    _Top = Canvas.GetTop(_CurrentItem);
    _OldRow = _GridHelper.GetRow(_CurrentItem);
    _OldColumn = _GridHelper.GetColumn(_CurrentItem);
```

```
      _CellItem = _GridHelper.GetItemByPozition(_OldRow, _OldColumn);
      _CurrentRow = MathCGAL.RoundDown((_Top + (_GridHelper.HeightCell / 2) + e.VerticalChange)
          / _GridHelper.HeightCell);
      if (_CurrentRow + _CellItem.CurrentItem.RowSpan > _GridHelper.CountRows)
      {
        _CurrentRow = _OldRow;
      }
      if (_OldRow != _CurrentRow)
      {
        _Changed = true;
        Canvas.SetTop(_CurrentItem, _CurrentRow * _GridHelper.HeightCell);
      }
      _CurrentColumn = MathCGAL.RoundDown((_Left + (_GridHelper.WidthCell / 2) + e.
          HorizontalChange) / _GridHelper.WidthCell);
      if (_CurrentColumn + _CellItem.CurrentItem.ColumnSpan > _GridHelper.CountColumns)
      {
        _CurrentColumn = _OldColumn;
      }
      if (_OldColumn != _CurrentColumn)
      {
        _Changed = true;
        Canvas.SetLeft(_CurrentItem, _CurrentColumn * _GridHelper.WidthCell);
      }
      if (_Changed)
      {
        _CellItem.CurrentItem.Row = _CurrentRow < 0 ? 0 : _CurrentRow;
        _CellItem.CurrentItem.Column = _CurrentColumn < 0 ? 0 : _CurrentColumn;
        _GridHelper.SetItemByPozitionForMove(_OldRow, _OldColumn, _CellItem.CurrentItem);
      }
    }
  }
```

Výpis 16: Algoritmus odchycení a zpracování události *DragDelta* pro změnu pozice ***master*** modulu

C.5 Algoritmus zpracování změny pozice *master* modulu

```
  public void SetItemByPozitionForMove(int? p_OldRow, int? p_OldColumn, CGModuleItem p_Item)
  {
    if (p_Item != null)
    {
      bool canChanged = true;
      List<CGModuleItem> changeItems = null;
      CGCellItem[,] matrixClone = (CGCellItem[,])_Matrix.Clone();
      CGCellItem findItem;
      changeItems = GetModuleItems(p_Item);
      if (changeItems.Count > 0)
      {
        SetItemToMatrix(matrixClone, p_OldRow.Value, p_OldColumn.Value, p_Item.RowSpan, p_Item.
            ColumnSpan, null);
        SetItemToMatrix(matrixClone, p_Item.Row, p_Item.Column, p_Item.RowSpan, p_Item.
            ColumnSpan, p_Item);
```

```
CGCellItem startPozition;
foreach (CGModuleItem item in changeItems)
{
  SetItemToMatrix(matrixClone, item.Row, item.Column, item.RowSpan, item.ColumnSpan,
      null, item);
  if (IsItemByPozitionEmpty(matrixClone, item.Row, item.Column))
  {
    startPozition = new CGCellItem(){ Column = item.Column, Row = item.Row };
  }
  else
  {
    startPozition = FindStarPozition(matrixClone, item.Row, item.Column);
  }
  if ( startPozition == null)
  {
    canChanged = false;
    break;
  }
  if (IsItemByPozitionEmpty(matrixClone, startPozition.Row, startPozition.Column, item.
      RowSpan, item.ColumnSpan))
  {
    findItem = GetItemByPozition(matrixClone, startPozition.Row, startPozition.Column);
  }
  else
  {
    findItem = GetFirstEmptyCell(matrixClone, startPozition.Row, startPozition.Column, item.
        RowSpan, item.ColumnSpan, null);
  }
  if (findItem == null)
  {
    findItem = GetFirstOptimalItem(matrixClone, GetEmptyMatrixUpRightCorner(matrixClone,
        startPozition.Row, startPozition.Column), item, startPozition.Row, startPozition.
        Column, startPozition.Row);
    if (findItem == null)
    {
      findItem = GetFirstEmptyCellForResize(matrixClone, startPozition.Row, startPozition.
          Column, item);
    }
  }
  else
  {
    findItem.ColumnSpan = item.ColumnSpan;
    findItem.RowSpan = item.RowSpan;
  }
  if (findItem != null)
  {
    item.RowNew = findItem.Row;
    item.ColumnNew = findItem.Column;
    item.ColumnSpanNew = findItem.ColumnSpan;
    item.RowSpanNew = findItem.RowSpan;
    MoveModuleToRightAndBottom(findItem, item, p_Item, matrixClone);
    SetItemToMatrix(matrixClone, item.RowNew, item.ColumnNew, item.RowSpanNew, item.
        ColumnSpanNew, item);
  }
```

```
            else
            {
              canChanged = false;
              break;
            }
          }
        }
        if (canChanged)
        {
          if (changeItems == null || changeItems.Count == 0)
          {
            SetItemToMatrix(_Matrix, p_OldRow.Value, p_OldColumn.Value, p_Item.RowSpan, p_Item.
                ColumnSpan, null);
            double width = p_Item.ColumnSpan * WidthCell;
            double height = p_Item.RowSpan * HeightCell;
            p_Item.CurrentModule.SetOptimalState(height, width);
            SetItemToMatrix(_Matrix, p_Item.Row, p_Item.Column, p_Item.RowSpan, p_Item.
                ColumnSpan, p_Item);
          }
          else
          {
            _Matrix = matrixClone;
            foreach (CGModuleItem item in changeItems)
            {
              item.Row = item.RowNew;
              item.Column = item.ColumnNew;
              item.ColumnSpan = item.ColumnSpanNew;
              item.RowSpan = item.RowSpanNew;
              item.RowNew = item.ColumnNew = item.ColumnSpanNew = item.RowSpanNew = -1;
              Grid.SetRow(item.ContentControl, item.Row);
              Grid.SetRowSpan(item.ContentControl, item.RowSpan);
              Grid.SetColumn(item.ContentControl, item.Column);
              Grid.SetColumnSpan(item.ContentControl, item.ColumnSpan);
              AddStartSocialInfoItem(item.Row, item.Column, item.RowSpan, item.ColumnSpan, item.
                  ModuleGuidDll, item.ModuleGuid, null);
              SetSizeAndCanvas(item, false);
            }
          }
          Grid.SetRow(p_Item.ContentControl, p_Item.Row);
          Grid.SetRowSpan(p_Item.ContentControl, p_Item.RowSpan);
          Grid.SetColumn(p_Item.ContentControl, p_Item.Column);
          Grid.SetColumnSpan(p_Item.ContentControl, p_Item.ColumnSpan);
          AddStartSocialInfoItem(p_Item.Row, p_Item.Column, p_Item.RowSpan, p_Item.ColumnSpan,
              p_Item.ModuleGuidDll, p_Item.ModuleGuid, null);
        }
        else
        {
          p_Item.Row = p_OldRow.Value;
          p_Item.Column = p_OldColumn.Value;
          SetSizeAndCanvas(p_Item, true);
        }
      }
    }
```

Výpis 17: Algoritmus zpracování změny pozice ***master*** modulu

C.6 Algoritmus odchycení a zpracování události *DragDelta* pro změnu velikosti *master* modulu

```
case HorizontalAlignment.Left:
{
  _HorizontalWidth = Math.Min(e.HorizontalChange,
  _CurrentItem.ActualWidth − _CurrentItem.MinWidth);
  if (e.HorizontalChange < 0)
  {
    _Added = MathCGAL.RoundUp((decimal)
    (Math.Abs(_HorizontalWidth) / _GridHelper.WidthCell));
    if (_Added > 0)
    {
      if (_CurrentColumn <= _Added)
      {
        _Added = _CurrentColumn;
        _CurrentColumn = 0;
        _CurrentColumnSpan += _Added;
      }
      else
      {
        _CurrentColumn −= _Added;
        _CurrentColumnSpan += _Added;
      }
      _Changed = true;
      Canvas.SetLeft(_CurrentItem, _CurrentColumn ∗ _GridHelper.WidthCell);
      _CurrentItem.Width = _CurrentColumnSpan ∗ _GridHelper.WidthCell − 13;
    }
  }
  else if (e.HorizontalChange > 0)
  {
    if (_CurrentColumnSpan > 1)
    {
      if (_HorizontalWidth > _GridHelper.WidthCell)
      {
        _Removed = Math.Abs(MathCGAL.RoundUp((decimal)
        ((_HorizontalWidth − _GridHelper.WidthCell) / _GridHelper.WidthCell)));
        if (_Removed > 0)
        {
          _CurrentColumn += _Removed;
          _CurrentColumnSpan −= _Removed;
          _Changed = true;
          Canvas.SetLeft(_CurrentItem, _CurrentColumn ∗ _GridHelper.WidthCell);
          _CurrentItem.Width = _CurrentColumnSpan ∗ _GridHelper.WidthCell − 13;
        }
      }
    }
  }
```

```
        break;
    }
```

Výpis 18: Algoritmus odchycení a zpracování události *DragDelta* pro změnu velikosti ***master*** modulu

D Reflexe

Základním zaveditelným prvkem aplikace pro *.NET framework* je *Assembly*(Sestava). *Assembly* mimo to také tvoří základní jednotku pro správu verzí, jednotné rozlišování typů a specifikaci bezpečnostních oprávnění.

Každá aplikace pro *.NET framework* je tvořena přinejmenším jednou *assembly* a ta je zase tvořena čtyřmi částmi, kterými jsou:

- *manifest*, který obsahuje *metadata* o *assembly*
- *metadata* o typech obsažených v *assembly*
- kód v jazyce *MSIL*, který je spuštěn prostředím *.NET runtime*
- zdroje(*resources*)

Metadata o obsažených typech a *MSIL* kód spolu tvoří takzvaný modul, což je přenositelný spustitelný (*PE - portable executable*) soubor. Nejjednodušší *assembly* jsou složeny z *manifestu* a jediného modulu s typy aplikace. I když to není časté, tak je možné vytvořit i sestavu s více moduly. Jednotlivé moduly s typy jsou pak představovány soubory s příponou *.netmodule*. Takovéto *assembly* jsou většinou tvořeny z optimalizačních důvodů, protože prostředí *.NET runtime* nahrává moduly pouze v případě potřeby typu v nich obsažených.

Metadata jsou data, která nesou popisné informace o *assembly* či typu. Například *manifest* obsahuje mimo jiné tato *metadata*:

- jednoduchý název *assembly*
- číslo verze *assembly*
- veřejný klíč tvůrce a *hešový* kód *assembly* (volitelně)
- seznam souborů, které tvoří danou *assembly* a jejich *hešové* kódy
- seznam typů, které tvoří *assembly* a informaci ke kterému modulu v *assembly* je konkrétní typ připojen

Jedním z důsledků použití mechanismu *metadat* pro všechny typy v prostředí *.NET frameworku* je možnost tyto typy v našem programu prozkoumávat a tato věc je nazývána **reflexe**. Jmenný prostor, který obsahuje nemalý počet tříd, jež námi mohou být použity pro manipulaci s danými elementy konkrétní aplikace nese název *System.Reflection*.

Mechanismus **reflexe** nám tedy umožňuje procházení a manipulaci s objektovým modelem představující konkrétní aplikaci. *Metadata*, která jsou **reflexí** využívána, jsou obvykle vytvářena kompilátorem při překladu aplikace. Kromě tohoto obvyklého způsoby tvorby *metadat* k elementům aplikace, je možné *metadata* vytvořit pomocí tříd, které se nacházejí ve jmenném prostoru *System.Reflection.Emit*.

Každá aplikace pro *.NET framework* běží v nějaké aplikační doméně, představující izolované běhové prostředí. Aplikační doménu je možné brát jako obdobu procesu ze světa programování ve *Win32*. Aplikační doména sama o sobě není popsána *metadaty*. Aplikační doména je představována třídou *AppDomain*, která nám pomocí své statické vlastnosti *CurrentDomain* předloží přístup k aktuální aplikační doméně aplikace a pokud se na ní zavolá metoda *GetAssemblies* jsou získány v podobě pole všechny instance třídy *Assembly*, která reflektuje objekt *assembly* v aplikační doméně. A stejně jako třída *Assembly* reflektuje *assembly* v aplikační doméně, tak ve jmenném prostoru *System.Reflection* existují i další třídy, které reflektují ostatní elementy aplikace *.NET*.[13]